LSTM Implementation

Bennett James

Abstract

In one of my master's classes, ECE564, which I took as a first semester senior year while in the Accelerated Bachelor's and Master's program at NC State. I was tasked with implementing an LSTM specifically the g(t) gate. This entails implementing functions for matrix multiplication, tanh, as well as both reading and writing to memory. To implement this, I separated my hardware into 3 stages. The first consisted of reading data from weight values and input values while matrix multiplying using a Design Ware multiplier. Once my accumulations were finished, they would go into a tanh function that would grab tanh values based upon inputs and interpolate data for maximum accuracy. After interpolations were completed the data was put stored in SRAM and the testbench saved the values in a g_result.dat file.

1. Introduction

The hardware being implemented was an LSTM specifically the g(t) gate. The function of this hardware is to implement a part of a LSTM (Long Short Term Memory Cell). The particular part I have implemented is the g(t) gate which consists of performing matrix multiplication of a 16x16 weight matrix by a 16x1 input and then performing the tanh function of this value. To implement this I separated my hardware into 3 stages. The first consisted of reading data from weight values and input values while matrix multiplying using a Design Ware multiplier. Once my 16 accumulations they would go into a tanh function that would grab tanh values based upon inputs and interpolate data for maximum accuracy. After interpolations are completed the data was put stored in SRAM and the test bench saved the values in a g_result.dat file.

The results of my design will be explained in more detail later in this report but as an overview I was able to pass all test cases in the demo files using modelsim as well as synthesize my design using synopsys2017 without having any major warnings or errors.

Throughout this report I plan to go over the micro-architecture and dive into the way I implemented my matrix multiplication function, my tanh function, and my data path. Then will touch on the interface between modules and how I connected the different functions together. The report will end with verification methods used as well as all results.

2. Micro-Architecture

To implement the high-level functions that are needed for the g(t) you need to be able to perform both matrix multiplication and the tanh function while reading in data. The most important aspect to implementing these functions is sign extending your registers to line up the decimal point in your number before performing calculations.

i. Reading and Writing

Accessing memory is very important in the function of this hardware. With regards to reading, I had to design hardware to read the weight values, the inputs, and the tanh values. To maximize efficiency in my design during my accumulate I was already referencing the next value to be read in for the next clock cycle as the previous inputs were multiplied and stored to eliminate any burned clock cycles that would result from using more states in my FSM.

For writing it was a simpler design as the only thing you had to do was write the output values to the correct address in SRAM but I had ensure I had pulled the write enable signal high at least one clock cycle prior to attempting to store data and to reset the write address in between rounds.

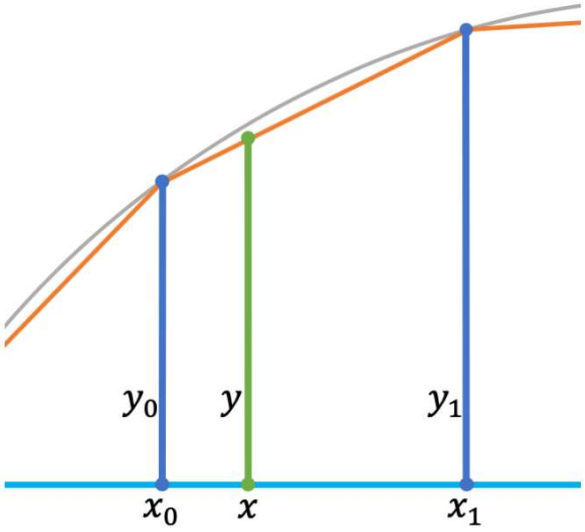
ii. Matrix Multiplication

To do this I performed matrix multiplication of a 16x16 matrix with a 16x1 matrix which would output a 16x1. This would need 256 multiplications to complete and there would be 16 outputs, and this would occur 16 times to iterate through the whole input file which contains 256 input values. When performing multiplications, I performed it by multiplying down the columns vertical without having to change the input value but the changing the direction of the output after each multiplication. The two inputs were 16 bit signed fraction numbers in which I was given a 32 bit number. With bits [31:30] being sign bits, bit 30 was dropped and the result was then accumulated with the current value in the respective accumulation register. The accumulation registers were 35 bits to allow for bit 34 to be the sign bit, bits [33:32] were used to check for saturation, [31:30] were integer bits and the fractions bits were [29:0]. When accumulations were done the value would be checked for saturation as for this project is a value was +/- 3.984375 the output was only marginally different. When al 16 accumulations were complete the done signal would go high causing the controller to stop sending the run signal and the run signal for tanh would go high.

iii. Tanh function

To implement this function, I was given a file containing 16-bit values for tanh inputs between 0 and 3.984375, these tanh values were only accurate up to the 6^{th} fraction bit while my outputs from matrix multiplication were accurate up until the 29^{th} fraction bit. To start the calculation, I would look at the sign bit of my input and convert the number to an unsigned binary if necessary while saving the sign bit for conversion back after the tanh value is calculated. To maximize accuracy when getting tanh values from the respective file, I would reference in memory the value that was accurate to my inputs first 8 most significant bits and this would be the lower bound from here I would save the value one above the as the upper bound as long as the input was not saturated in which the next step would be skip and the lower bound would be directly outputted. After receiving both the high and low bounds. Linear interpolation would be performed to

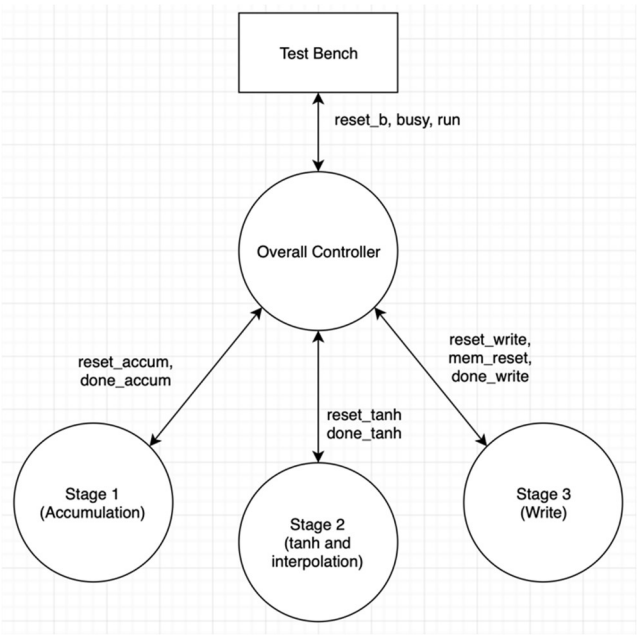
maximize accuracy. The input to the tanh function initially was a 32 bit number therefore the addresses that were used in the lookup table were sign extended to 32 bit with emphasis on the placement of these bits to account for the decimal point. After performing the function below a 64 bit unsigned number was generated. From here bits [59:45] of this calculated value were assigned to an output reg after sign correction was performed the sign corrected value would then be stored on the same registers, I stored the accumulations on previously. This was performed for all 16 values and when finished a done signal was sent to signal the controller to begin writing these values to memory.



$$y = \frac{y_0(x_1 - x)}{x_1 - x_0} + \frac{y_1(x - x_0)}{x_1 - x_0}$$

iv. High Level Architecture

My design was initialized when the testbench pulled down the reset signal which was fed directly into my controller module and then shortly after would increment the run cycle high for one clock cycle. From here the controller would put the busy signal high to signal to the testbench calculations were being performed. Then the controller would initialize each stage as necessary based upon several variables and counters used to track the progress of the calculations. A diagram of the high-level architecture can be seen below. Once completed the busy signal would be dropped and the testbench would either put the run signal high again causing another round to begin or finish the simulation.



v. The Datapath

After the controller was initialized and the first stage of the hardware was entered which contained the reading and accumulating. The data for the weights and inputs would be stored on respective wires and put into a DesignWare multiplier this output value would be fed into the accumulation module to be accumulated with the previous accumulation values and the number of accumulations done were tracked to signal when complete. This was done 256 times as this is the number of multiplications required to complete a 16x16 times 16x1 matrix multiply. From this stage the data would next be used in the tanh stage. Within tanh the data would be used to address the high and low values known in memory and then interpolated using the methods described in section 2.ii. Once tanh had been completed on the current set of 16 values, we were dealing with the write run signal would be put high to allow for these values to be written to memory. Once this was complete a counter would be incremented this counter was used in a condition statement that waited for the total number of run to be equal to 16 which would signal, we had written all 256 inputs and the calculations we were asked to perform were complete. Signaling the controller to lower the busy signal and relieve control to the testbench to signal the next action.

3. Interface Specification

From top level the test bench only had a few interactions with my design. The test bench would send a reset signal and a run signal, and my design would output a busy signal to the testbench. The reset signal was sent on initialization and the run signal was sent in between “rounds” which consisted of 256 inputs each. Below is a full list of registers and wires used in my design both to interact between modules and internally.

Controller Signal Name/Width (bits)	Type of signal, connecting module	Function
run, reset (1)	Input, testbench	Reset is pulled low for one clock cycle on initialization and run is high for one clock cycle to start a new round
Busy (1)	Output, testbench	Tells testbench that it is still working
Done signals (accum, tanh, and write)	Input, lower level FSMs	Received signals from each other FSM signaling to go to next step
Control signals (accum, tanh, and write) (1)	Output, lower level FSMs	Controls each lower level FSM and whether or not it should run
Reset signal (accum, tanh, write) (1)	Output, lower level FSMs	Resets each FSM based upon conditions for reset whether in between sets of 16 inputs or rounds (256 inputs)
Mem_reset (1)	Output, read module	When run signal from testbench is received this signal goes high

Read Module Signal Name/Width (bits)	Type of signal, connecting module	Function
run, reset, full_reset (1)	Input, controller	Comes from the controller to reset/run read
num_of_accumuation (8)	Input, accumulation	Keeps track of number of accumulations completed
stage_done (1)	Output, controller	Once 256 accumulations stage is complete
sram_read_address, g_read_address (12)	Output, testbench	Addresses in memory we are requesting values from
row_increment, col_increment (9)	Internal	Used to keep track of the memory locations we were reading from
state, next_state (3)	Internal	Keeps track of state for the FSM controlling the read module

Accumulation Module Signal Name/Width (bits)	Type of signal, connecting module	Function
run, reset (1)	Input, controller	Run and reset signals from controller
out_mult (16)	Input, DW02_Mult	Output from the DW02_mult
num_of_accumulations (8)	Output, accumulation	Tracks number of accumulations complete
current_output (33)	Output, modulelinks	Determined by mult_select and is stored in global reg
done_vals (4)	Output, controller	Determines which global reg the current_output is stored
saturation_temp (35)	Internal	Reg with 2 extra bits to check for saturation
mult_select (4)	Internal	Selects where we are adding the current value of out_mult too
initialize_flag (1)	Internal	Initialized all necessary outputs and registers

Tanh Signal Name/Width (bits)	Type of signal, connecting module	Function
run, reset (1)	Input, controller	Run and reset signals from controller
tanh_in_lookup (33)	Input, modulelinks	Value coming in from accumulation
tanh_mem_val_in (16)	Input, testbench	Value incoming from tanh address
tanh_mem_addr (12)	Output, testbench	Address referencing in memory
interpolated_val (15)	Output, tanh sign correct	Interpolated value which goes to check for sign correction
tanh_sign_out (1)	Output, tanh sign correct	Sign of accumulated in value and goes out with interpolated value
interpolations_done (4)	Output, modulelinks	When an interpolation is done selects next value to be interpolated.
ready_for_write (4)	Output, modulelinks	When an interpolation is done selects where to store the current value
stage_done (1)	Output, controller	Signal controller are tanh calculations are complete
interpolated_val_temp (64)	Internal	Number containing the sum of the 2 im_vals
state, next_state (4)	Internal	States for FSM
unsigned_tanh_lookup (32)	Internal	Unsinged value of the input value
tanh_mem_addr_hi (32)	Internal	Address of the high value for tanh lookup
Tanh_mem_addr_lo (32)	Internal	Address of the low value for tanh lookup
extended_tanh_mem_addr_lo (32)	Internal	Sign extended high address with emphasis on decimal point
extended_tanh_mem_addr_hi (32 bits)	Internal	Sign extended low address with emphasis on decimal point
bit_extended_tanh_hi (32)	Internal	Bit extended tanh high value
bit_extended_tanh_lo (32)	Internal	Bit extended tanh low values
im_val1,2 (64)	Internal	Calculations were intermediary multiplication signals

Tanh sign correction Signal Name/Width (bits)	Type of signal, connecting modules	Function
unsigned_tanh_in (15)	Input, tanh	From tanh module value is passed here for sign correction
tanh_sign_in (1)	Input, tanh	Determines if sign correction is needed
signed_tanh_out (16)	Output, modulelinks	Correct signed value is outputted here

Write Signal Name/Width (bits)	Type of signal, connecting modules	Function
run, reset, full_reset (1)	Input, controller	Runs and resets write module based upon controller
write_select (4)	Output, moduleslinks	Selects which modulelink value we are writng to memory
write_address (12)	Output, testbench	Address in memory where we are storing data
write_enable (1)	Output, testbench	Must be enabled before data can be written
stage_done (1)	Output, controller	Signaled once all necessary data for current stage is complete
next_state, state (3)	Internal	States for FSM

4. Technical Implementation

At a high level my hardware contains a main controller and 3 lower level controllers depending on the stage of calculations. When modeling my controller my main point I had to base it around was referencing memory because depending on which calculations we were doing, and the time needed to gather all the data from memory. This would change throughout each stage as for multiplication data could be continuously read, multiplied, and stored as while for the tanh function you needed to read two addresses before making one calculation. My first stage reads and accumulated it is streamlined to allow for when a value is received the next value is already being requested maximizing efficiency and for this stage to be completed as quickly as possible. The second stage was tanh, this was a more complex stage and could not be as streamlined due to having to reference two points in memory before performing the first calculation. To maximize timing here the FSM I implemented saves a value and requests the next value simultaneously to allow for the second received to be received on the next clock edge. The third and final stage was write, this stage was done in a very similar fashion to the first stage where we were reading from memory. It was streamlined and each high level register was referenced and outputted while incrementing the memory register to have the next address ready next clock cycle to directly reference the next value. Once all three stages were completed and write had sent its done signal to the controller the controller would check the total number of inputs we had iterated over if we had not iterated over all 256 inputs then the FSM would be reset while saving the memory addresses were next read and save points to therefore implement through again and save those values in their respective addresses. Once all 256 outputs had been written the high level controller would lower the busy signal and the test bench would take over by either sending another run signal to perform another round of calculations or finish the simulation in which the accuracy of your gate could be seen.

5. Verification

When testing my g(t) gate, I used a test bench that would compare my results against a file containing the expected results. The test bench measured accuracy up to a tolerance of 0.0039 and would also state what your accuracy would be if the tolerance were expanded to 0.0078. Both the expected results and my outputs were in .dat files and consisted of 16 bits. My numbers were accurate to the 16^{th} bit in most cases with the only difference being the rounding of LSB.

6. Results Achieved

After running my design through the testbench, my design met 0.0039 tolerance that was set by the testbench for all inputs. When synthesizing my design through synopsys I was able to achieve a clock cycle of 1.4 ns, an area of 15786.568 um^2 and a total power usage of 5.5274 mW.

7. Conclusions

After completing this project, I have gained an understanding behind the implementation of an LSTM even though I only implemented the g(t) gate. I have successfully implemented both matrix multiplication and a tanh function. My results were accurate of to the 16^{th} bit with the expected results and the testbench stated I had 100% accuracy with my results due to the tolerances it had set which was 0.0039. With regards to an LSTM the only other large functions are sigmoid functions and Hadamard product. Though the full LSTM is much more complex with many parts integrating together this was a base that from here I could with adequate time design and implement the full LSTM.

Appendix

Timing Report

```
Point                                       Incr       Path
-------------------------------------------------------------
clock clk (rise edge)                     0.0000     0.0000
clock network delay (ideal)               0.0000     0.0000
W5_state_reg_0_/CK (DFFR_X1)              0.0000     0.0000 r
W5_state_reg_0_/QN (DFFR_X1)              0.6690     0.6690 r
U7675/ZN (OAI21_X2)                       0.1513     0.8203 f
W5_state_reg_0_/D (DFFR_X1)               0.0000     0.8203 f
data arrival time                                    0.8203

clock clk (rise edge)                     1.4000     1.4000
clock network delay (ideal)               0.0000     1.4000
clock uncertainty                        -0.0500     1.3500
W5_state_reg_0_/CK (DFFR_X1)              0.0000     1.3500 r
library setup time                       -0.4424     0.9076
data required time                                   0.9076
-------------------------------------------------------------
data required time                                   0.9076
data arrival time                                   -0.8203
-------------------------------------------------------------
slack (MET)                                          0.0873
```

Area Report

```
Number of ports:                         2614
Number of nets:                         11986
Number of cells:                         8592
Number of combinational cells:           7081
Number of sequential cells:              1486
Number of macros/black boxes:               0
Number of buf/inv:                       1584
Number of references:                      73

Combinational area:              11610.102138
Buf/Inv area:                      844.550008
Noncombinational area:            4176.466163
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                 15786.568301
Total area:                 undefined
1
```

Power Report

```
                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000            0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000            0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000            0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000            0.0000  (   0.00%)
register       5.2807e-02       1.7449e-03        2.0779e+05        5.4759e-02  (   0.99%)
sequential         0.2596       2.3579e-02        2.4504e+09            2.7336  (  49.46%)
combinational      1.6255           1.0457        6.7921e+07            2.7391  (  49.55%)
--------------------------------------------------------------------------------------------------
Total              1.9379 mW        1.0710 mW     2.5185e+09 pW         5.5274 mW
```