https://youtu.be/hSjiDW68b34
https://github.com/bird-wizard/WES_237C/tree/main/bnn

BNN Design Report:

golden.h:
Due to time constraints and for easier debugging, I maintained golden.h to 10 samples and borrowed most logic from create_input() function to access mnist test data. After executing the feed forward quantized, I saved out the weights, inputs, and golden_output variables I wanted in golden.h using a lot of python file write commands.

bnn_test.cpp:
Once I had golden.h with inputs, weights, and golden output, I re-wrote the test bench to use the gathered data. I read in the inputs, use the weights for running BNN, and finally compare the output to the golden_outputs.

bnn.cpp
Due to time constraints and extensive debugging with HLS interface choices, I was not able to include optimizations to the bnn algorithm. XNOR and dot_xnor functions are directly from the textbook. The following layers of execution are a direct reflection of the feed_forward_quantized python function.

bnn.ipynb:
Now here's where I spent more time expanding the testing from 10 samples to all samples from the mnist test data. **Note:** at this point I realized I forgot to quantize the input at the bnn.cpp function level, so the input needed to be quantized before going through the PYNQ board, hence, the binarize function. On the first pass through, we predict only one sample using the predict_one() function call. In order to test subsequent values in a loop, test_batch function was born. Throw in some cool metrics at the end to tally up some data and we finally have a measurement on the hardware performance.

Conclusion:
With extra time, I would revisit and quantize the input at the bnn.cpp level. HLS optimizations would include bit-width optimization, cyclic array partitioning of the weights, and loop unrolling and pipelining of the dot_xnor and sign executions.