Variational Auto Decoder For Audio: A Chiplet Design Experiment

Jack Knopp

Abstract

This paper presents the design and implementation of a hardware-accelerated decoder for variational autoencoders (VAEs) targeted at real-time audio synthesis applications. A custom VAE network was initially designed and trained for bass drum sample generation using PyTorch, but the scope was reduced to a ”toy” proof-of-concept network due to the complexity of implementing large-scale neural networks in hardware using large language models (LLMs) as co-design agents. Strategies and issues related to LLM co-design are discussed throughout. The final implementation consists of a fully connected layer with ReLU activation, upsampling layers, and three 1D convolution layers, all designed in SystemVerilog with full parallelization using combinational logic blocks. The design employs a modular, bottom-up approach where each layer was individually implemented and verified before integration into a non-pipelined state machine. Despite resolution issues causing small mismatches in some calculations, the hardware implementation achieves a 20× speedup compared to the pure Python reference implementation when accounting for SPI interface overhead. The final design was synthesized using OpenLane with the SkyWater 130 nm PDK, resulting in a 13,644-device chip with a die size of 532 um × 542 um operating at 100 MHz. This work demonstrates the feasibility of implementing VAE decoders in dedicated hardware for real-time audio applications, with clear pathways for scaling to larger, more practical network dimensions.

1 Intro

Variational autoencoders are a generative architecture used in many disciplines including audio, video, NLP, and more. For audio VAEs, a dataset of waveforms are encoded into what is called a latent vector and then decoded back into the original waveform. To train the network, the loss of the recreated network is propagated back through the network in order for the decoded waveform to match the original as closely as possible. This is shown in the following figure.

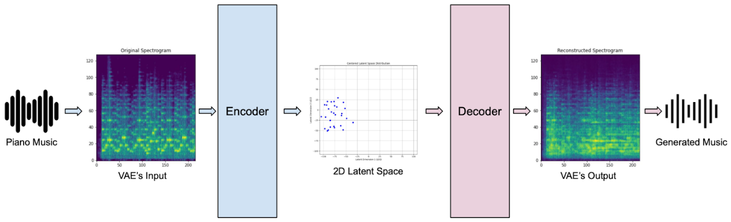


Figure 1: Flow chart of audio VAE

After training, one can then just pass random latent vectors through the decoder to create new waveforms that have the same underlying structure as the original dataset. This can be done with any type of dataset, but of particular interest to the author was drum samples. It would be of interest to create new drum samples in real time with a decoder network on dedicated hardware with the possibility of modulating the latent vector in real time.

The goal of the project discussed below is to implement a version of the decoder on an FPGA using LLMs as a software co-designer in order to accelerate a decoder network designed in Python.

2 Initial Design

Several weeks were spent trying to understand and recreate VAE (or VAE-like) networks such as Google's NSynth or RAVE. NSynth's autoregressive and therefore sequential nature does not lend itself well to parallelization and the accompanying performance speedups. RAVE on the other hand takes days to train. Neither were viable given the time frame of the project, so a custom VAE was designed with ChatGPT. The encoder network is not relevant for this discussion as only the decoder is to be implemented, but note that the waveform is first converted to mel spectrograms before encoding and needs to be converted back into the time domain. The decoder is a simple network and is comprised of a fully connected layer with a ReLU activation followed by upsampling and 3 1D convolutional layers, each with ReLU activations and the first two with 2× nearest neighbor upsampling. The output of this network results in a matrix of size 128 channels × 512 frames. This matrix is then resynthesized using an inverse mel-spectrogram algorithm to create or recreate the waveform.

This network was trained with a library of bass drum samples and although a simple network, it yielded fairly good recreation and interesting random latent space results. An example recreation of an input vector and random latent creation is shown below.

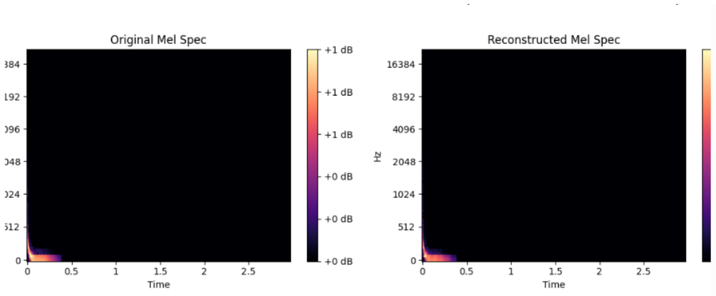


Figure 2: Initial BD waveform and recreated waveform

This experiment is better listened to than visualized, and results for both the recreation and random latent vector sampling can be found on the project GitHub in the Colab notebook `DRUM_VAE.ipynb`. The training process for this network is also explicitly defined in this file.

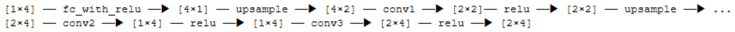
3 Reduction in Scope

It became apparent after several weeks of attempting to implement even a single large fully connected or convolution layer of the network discussed above in SystemVerilog with ChatGPT that it was beyond the abilities of the LLM. This is discussed more in the Design Strategy section, but there were a lot of initial syntax errors that proved to be insurmountable, particularly when using the Icarus simulator both locally and on EDAplayground. Given that the LLM has no internal simulator and that training code was used indiscriminately regardless of the development environment, it is of no surprise that ChatGPT struggled in this regard.

Given these challenges, there were two possible solutions for reducing the scope in order to finish within the time frame. Either only a single layer or several layers could be implemented, or a smaller version could be designed as a proof of concept. Due to the challenges seeming to stem from the size of the inputs and outputs of each layer, and because a SPI layer would need to be designed to take input AND output data rather than just output, a reduced ”toy” network was designed and implemented as a proof of concept. Given a working toy network, it should be of little consequence to scale up the dimensions, at least in terms of HDL complexity. A scaled-up network will of course increase the die area and power, and reduce the speed, but as a proof of concept this design is likely more valuable than just implementing a large single layer of the network.

4 Final (Toy) Network

The final toy network after the reduction in scope for use in the final project is shown below.



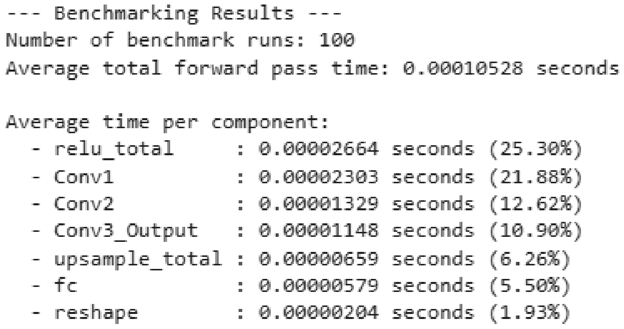
```
[1×4] — fc_with_relu —▶ [4×1] — upsample —▶ [4×2] — conv1 —▶ [2×2]— relu —▶ [2×2] — upsample —▶ ...
[2×4] — conv2 —▶ [1×4] — relu —▶ [1×4] — conv3 —▶ [2×4] — relu —▶ [2×4]
```

Figure 3:

It includes a fully connected layer with ReLU activation, an upsampling layer, followed by two layers of 1D convolution, ReLU, and upsampling, and a final 1D convolution layer with ReLU output. This is similar to the tested PyTorch network and will prove a good baseline that can be scaled up in either channels, frames, or dimensions. Implementing each individual layer and tying them together in a state machine has proven to be a difficult enough challenge.

4.1 Benchmarking

A pure Python (no torch, numpy, etc.) network of the architecture described above was created to benchmark the network on a CPU and identify the bottlenecks. To do this, a random latent vector was passed through the network 100 times and the average time through the network as well as the average time through each layer type was calculated. The benchmarking results are as follows.



```
--- Benchmarking Results ---
Number of benchmark runs: 100
Average total forward pass time: 0.00010528 seconds

Average time per component:
  - relu_total     : 0.00002664 seconds (25.30%)
  - Conv1          : 0.00002303 seconds (21.88%)
  - Conv2          : 0.00001329 seconds (12.62%)
  - Conv3_Output   : 0.00001148 seconds (10.90%)
  - upsample_total : 0.00000659 seconds (6.26%)
  - fc             : 0.00000579 seconds (5.50%)
  - reshape        : 0.00000204 seconds (1.93%)
```

Figure 4:

This was a bit of a surprise. The ReLU layer takes a quarter of the total time. Granted, nearly half the total layers are ReLUs, but they are a very simple operation, passing the value through if greater than zero else pass zero. Given the relative simplicity and size of all other layers however (even the convolution layers are only a few operations), this makes sense. If the number of channels and frames were to increase however, the convolution blocks would begin to dominate.

It is also of note that this was relatively slow overall, at just over 100 us per latent input. I would suspect little parallelization is taking place as no libraries are being used for the implementation. If this network were to be implemented in an optimized library like PyTorch, it would be of little surprise if there was a speedup even with the added overhead of an external library.

5 HDL Design

5.1 Overview

The following was done with extensive help from Google Gemini. ChatGPT was the initial choice for the co-design coding agent, but the scope of the project (the Verilog code ended up being around 1000 lines) proved too challenging for all OpenAI models, including o4-mini-high, which is a reasoning model built specifically for coding. There are a few tricks that were critical to achieving anything close to a (semi) completed design. These are discussed in the section below.

5.2 Design Strategy

5.2.1 Strategy 1: Modularization and Bottom-Up Design

Scope of a software design project has not typically been an issue (at least in the author's experience) with LLMs, provided that appropriate descriptions are given. In this case, the description given was a small neural network described in Python where inputs, outputs, sizes, and operations are explicitly defined. Historically, porting a design of this kind to a different language has not been an issue. However, this was not the case when porting to SystemVerilog. The reason for this seems twofold. First, the model cannot compile the design or run the testbench on its own. This requires an outside program (in our case, primarily edaplayground.com), and the debugging takes a lot of back and forth, with the majority of confusion from the LLM coming from not understanding the compiler or environment in which things are being built. Second (and this is speculation), there is likely more Python code in GitHub and other codebases that the LLMs were trained on than SystemVerilog. The more training examples, the better the output, and in general the LLMs had an easy time doing anything in Python and a very challenging time doing the same task in Verilog. To build this design, the LLM was requested to code each layer (fully connected, ReLU, Conv1, etc.) one at a time. When the layer was completed and smoke-tested with a testbench, it was added to the single design file being used, and development was started on the next layer. Once all layers were completed, the state machine was developed by adding one layer at a time and testing the partial network with another testbench. This eventually led to a completed network and better understanding from the author of what was expected at each layer.

5.2.2 Strategy 2: Keeping Task Within Context

Given the large scope of this design, it was important to keep the LLM focused on the requested design block within the context of the entire decoder. The most effective way to do this (without paying for the coding agent that works within GitHub or an entire file system) was to keep everything in a single file. It ended up being unwieldy and difficult to parse and is certainly not the preferred way to develop code from a human perspective. But this seemed to help Gemini understand the changes it was making relative to the rest of the network and how one change would impact something else in the code. The downside of this is that it takes minutes, sometimes 5+, to output the updated file. Compounding this issue was a seeming bug in Gemini where it would lose track of whether it was coding in markdown or direct output to the user, making copy-paste difficult. However, the combination of these strategies ended up resulting in a completed network. In order for the author to better understand the code, each layer is discussed below in detail.

5.3 Fully Connected with ReLU Layer

5.3.1 Design

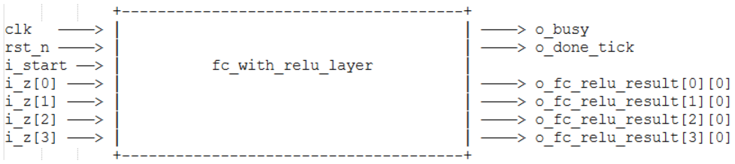


Figure 5: `fc_with_relu_layer`

This module is a fully connected layer $o\_fc\_relu\_result = \text{ReLU}(Wi\_z + bias)$, where ReLU is computed element-wise on the output. Note that $W \in \mathbb{R}^{O\times L}$ and $i\_z, o\_fc\_relu\_result, bias \in \mathbb{R}^{L\times 1}$ where O is the output dimension, and L is the latent vector dimension. Because the loop is defined within an `always_comb` block, this is a fully parallelized operation where each vector × vector happens in parallel as well as each vector element × element product happening in parallel. This is a 2-cycle operation which can be scaled up to accommodate larger input and output sizes, but for this toy example we'll use $W \in \mathbb{R}^{4\times 4}$ and $i\_z, o\_fc\_relu\_result, bias \in \mathbb{R}^{4\times 1}$. Note that for this and all future blocks 16 bit fixed point multiplication using shifts is used in addition to clipping in order to prevent overflow and underflow.

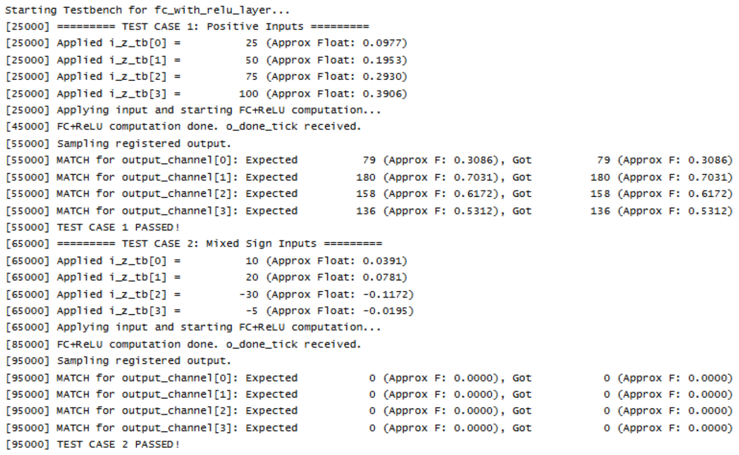
The state machine only has two states related to this layer: idle and compute. As one would expect, nothing is happening during idle. When the block receives `i_start = 1` it begins computing and completes computation outputting `o_busy = 1`. On the next cycle it outputs `o_tick = 1` indicating the computation is complete.

5.3.2 Verification

The testbench `tb_fc_with_relu_layer.sv` checks two inputs against known output values: one with mixed positive and negative inputs expecting only zero outputs given the weights, and one with only positive outputs given the weights. As seen below, both pass.

5.4 Fully Connected with ReLU Layer Verification

5.4.1 Design



```
Starting Testbench for fc_with_relu_layer...
[25000] ========= TEST CASE 1: Positive Inputs =========
[25000] Applied i_z_tb[0] =         25 (Approx Float: 0.0977)
[25000] Applied i_z_tb[1] =         50 (Approx Float: 0.1953)
[25000] Applied i_z_tb[2] =         75 (Approx Float: 0.2930)
[25000] Applied i_z_tb[3] =        100 (Approx Float: 0.3906)
[25000] Applying input and starting FC+ReLU computation...
[45000] FC+ReLU computation done. o_done_tick received.
[55000] Sampling registered output.
[55000] MATCH for output_channel[0]: Expected         79 (Approx F: 0.3086), Got         79 (Approx F: 0.3086)
[55000] MATCH for output_channel[1]: Expected        180 (Approx F: 0.7031), Got        180 (Approx F: 0.7031)
[55000] MATCH for output_channel[2]: Expected        158 (Approx F: 0.6172), Got        158 (Approx F: 0.6172)
[55000] MATCH for output_channel[3]: Expected        136 (Approx F: 0.5312), Got        136 (Approx F: 0.5312)
[55000] TEST CASE 1 PASSED!
[65000] ========= TEST CASE 2: Mixed Sign Inputs =========
[65000] Applied i_z_tb[0] =         10 (Approx Float: 0.0391)
[65000] Applied i_z_tb[1] =         20 (Approx Float: 0.0781)
[65000] Applied i_z_tb[2] =        -30 (Approx Float: -0.1172)
[65000] Applied i_z_tb[3] =         -5 (Approx Float: -0.0195)
[65000] Applying input and starting FC+ReLU computation...
[85000] FC+ReLU computation done. o_done_tick received.
[95000] Sampling registered output.
[95000] MATCH for output_channel[0]: Expected          0 (Approx F: 0.0000), Got          0 (Approx F: 0.0000)
[95000] MATCH for output_channel[1]: Expected          0 (Approx F: 0.0000), Got          0 (Approx F: 0.0000)
[95000] MATCH for output_channel[2]: Expected          0 (Approx F: 0.0000), Got          0 (Approx F: 0.0000)
[95000] MATCH for output_channel[3]: Expected          0 (Approx F: 0.0000), Got          0 (Approx F: 0.0000)
[95000] TEST CASE 2 PASSED!
```

Figure 6: `tb_fc_with_relu_layer` Verification

HOW TO TEST: *To test the module, go to https://www.edaplayground.com/, paste **`final.sv`** in **`design.sv`** and **`tb_fc_with_relu_layer.sv`** in **`testbench.sv`**, set simulator to Synopsys and run to see output.*

5.5 Upsampling Module

5.5.1 Design

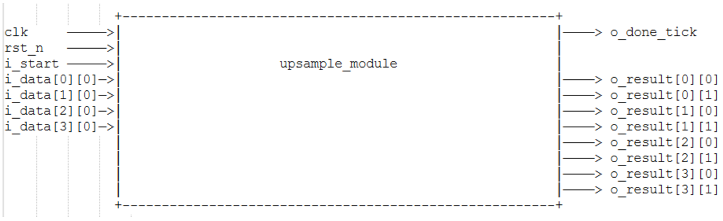


Figure 7: 1st `upsample_module` block

The upsampling module is used at the output of each RELU after the Fully Connected with ReLU Layer, and Convolution 1, and Convolution 2 blocks. It simply doubles the length of the number of columns in a matrix (or array) through nearest neighbor interpolation e.g. repeating each number in each row. For example an array `i_data = [1,2], [3,4]` would turn into `o_result = [1,1,2,2], [3,3,4,4]`. This operation is completed for all rows and the module parameterized to be used at in all three upsample blocks.

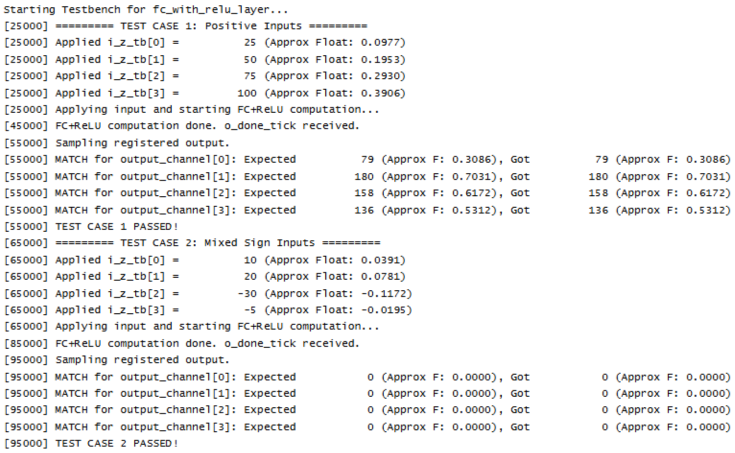
Similarly to the Fully Connected with Relu implementation, this also uses one big `always_comb` block for full parallelization across all elements. It uses a similar strategy for the state machine with two states idle, and compute. When the block receives `i_start = 1` it begins computing and completes computation outputting `o_busy = 1`. On the next cycle it outputs `o_tick = 1` indicating the computation is complete.

5.5.2 Verification

The testbench `tb_upsample_module.sv` again checks two inputs against known output values: one with mixed positive and negative inputs and one with only positive values. As seen below both successfully upsample the signal.

5.6 Upsample Verification

5.6.1 Design



```
Starting Testbench for fc_with_relu_layer...
[25000] ========= TEST CASE 1: Positive Inputs =========
[25000] Applied i_z_tb[0] =          25 (Approx Float: 0.0977)
[25000] Applied i_z_tb[1] =          50 (Approx Float: 0.1953)
[25000] Applied i_z_tb[2] =          75 (Approx Float: 0.2930)
[25000] Applied i_z_tb[3] =         100 (Approx Float: 0.3906)
[25000] Applying input and starting FC+ReLU computation...
[45000] FC+ReLU computation done. o_done_tick received.
[55000] Sampling registered output.
[55000] MATCH for output_channel[0]: Expected          79 (Approx F: 0.3086), Got          79 (Approx F: 0.3086)
[55000] MATCH for output_channel[1]: Expected         180 (Approx F: 0.7031), Got         180 (Approx F: 0.7031)
[55000] MATCH for output_channel[2]: Expected         158 (Approx F: 0.6172), Got         158 (Approx F: 0.6172)
[55000] MATCH for output_channel[3]: Expected         136 (Approx F: 0.5312), Got         136 (Approx F: 0.5312)
[55000] TEST CASE 1 PASSED!
[65000] ========= TEST CASE 2: Mixed Sign Inputs =========
[65000] Applied i_z_tb[0] =          10 (Approx Float: 0.0391)
[65000] Applied i_z_tb[1] =          20 (Approx Float: 0.0781)
[65000] Applied i_z_tb[2] =         -30 (Approx Float: -0.1172)
[65000] Applied i_z_tb[3] =          -5 (Approx Float: -0.0195)
[65000] Applying input and starting FC+ReLU computation...
[85000] FC+ReLU computation done. o_done_tick received.
[95000] Sampling registered output.
[95000] MATCH for output_channel[0]: Expected           0 (Approx F: 0.0000), Got           0 (Approx F: 0.0000)
[95000] MATCH for output_channel[1]: Expected           0 (Approx F: 0.0000), Got           0 (Approx F: 0.0000)
[95000] MATCH for output_channel[2]: Expected           0 (Approx F: 0.0000), Got           0 (Approx F: 0.0000)
[95000] MATCH for output_channel[3]: Expected           0 (Approx F: 0.0000), Got           0 (Approx F: 0.0000)
[95000] TEST CASE 2 PASSED!
```

Figure 8: `tb_upsample_module.sv` Verification

HOW TO TEST:*To test the module, go to https://www.edaplayground.com/, paste* `final.sv` *in* `design.sv` *and* `tb_upsample_module.sv` *in* `testbench.sv`*, set simulator to Synopsys and run to see output.*

5.7 Convolution Layers

All three convolution layers work similarly to one another with differences in expected input and output sizes, kernel sizes, and stride sizes. Note that there are three separate modules rather than one generic one. This was easier for the LLM to complete. These are 1D convolution blocks that compute a standard kernel × input multiplication and stride shift with zero padding for size control. All Conv blocks are also fully parallelized with each kernel × input operation being completed at the same time using a single `always_comb` block as previously seen. With this parallelization, each block takes two frames to compute.

5.7.1 Conv1

This block has an input size of 4×2, and with a stride size of 1, zero padding of 1, and kernel of 3, results in a 2×2 output. The figure shows the block diagram for this module.

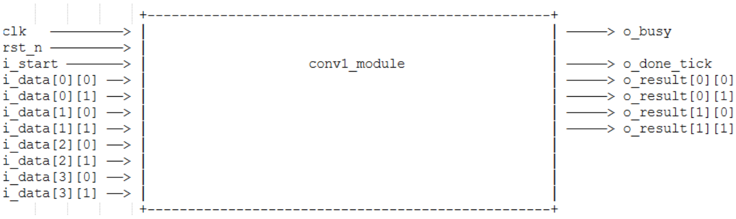
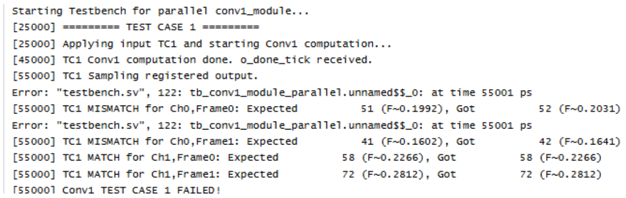


Figure 9: `conv1` module

The testbench `tb_conv1_module.sv` again checks one input against known output values. As seen below, we have small one-bit mismatches in two of the values likely due to resolution. Overall, it is close enough.



```
Starting Testbench for parallel conv1_module...
[25000] ========= TEST CASE 1 =========
[25000] Applying input TC1 and starting Conv1 computation...
[45000] TC1 Conv1 computation done. o_done_tick received.
[55000] TC1 Sampling registered output.
Error: "testbench.sv", 122: tb_conv1_module_parallel.unnamed$$_0: at time 55001 ps
[55000] TC1 MISMATCH for Ch0,Frame0: Expected          51 (F~0.1992), Got          52 (F~0.2031)
Error: "testbench.sv", 122: tb_conv1_module_parallel.unnamed$$_0: at time 55001 ps
[55000] TC1 MISMATCH for Ch0,Frame1: Expected          41 (F~0.1602), Got          42 (F~0.1641)
[55000] TC1 MATCH for Ch1,Frame0: Expected          58 (F~0.2266), Got          58 (F~0.2266)
[55000] TC1 MATCH for Ch1,Frame1: Expected          72 (F~0.2812), Got          72 (F~0.2812)
[55000] Conv1 TEST CASE 1 FAILED!
```

Figure 10: `tb_conv1_module.sv` Verification

5.7.2 Conv2

After upsampling, the input to this block has a size of 4×2, with a stride size of 1, zero padding of 1, and kernel of 3, which results in a 4×1 output. The figure shows the block diagram for this module.

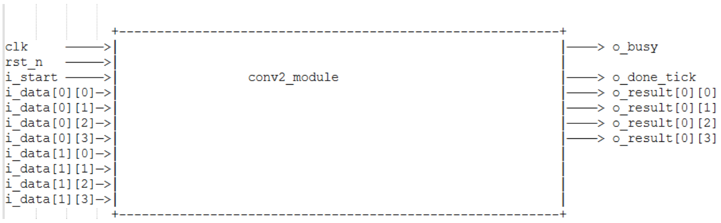
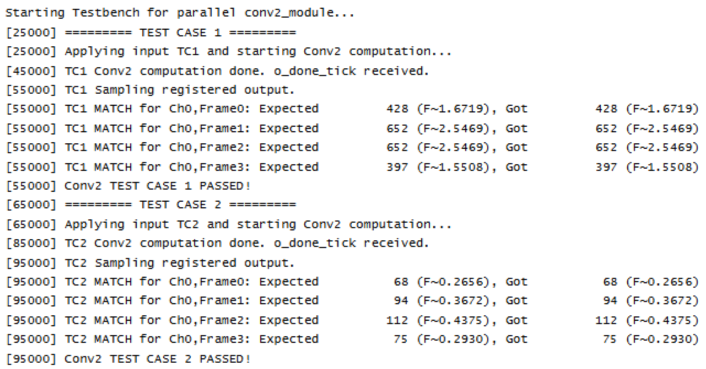


Figure 11: `conv2` module

The testbench `tb_conv2_module.sv` again checks two inputs against known output values. As seen below, both tests pass.



```
Starting Testbench for parallel conv2_module...
[25000] ========= TEST CASE 1 =========
[25000] Applying input TC1 and starting Conv2 computation...
[45000] TC1 Conv2 computation done. o_done_tick received.
[55000] TC1 Sampling registered output.
[55000] TC1 MATCH for Ch0,Frame0: Expected        428 (F~1.6719), Got        428 (F~1.6719)
[55000] TC1 MATCH for Ch0,Frame1: Expected        652 (F~2.5469), Got        652 (F~2.5469)
[55000] TC1 MATCH for Ch0,Frame2: Expected        652 (F~2.5469), Got        652 (F~2.5469)
[55000] TC1 MATCH for Ch0,Frame3: Expected        397 (F~1.5508), Got        397 (F~1.5508)
[55000] Conv2 TEST CASE 1 PASSED!
[65000] ========= TEST CASE 2 =========
[65000] Applying input TC2 and starting Conv2 computation...
[85000] TC2 Conv2 computation done. o_done_tick received.
[95000] TC2 Sampling registered output.
[95000] TC2 MATCH for Ch0,Frame0: Expected         68 (F~0.2656), Got         68 (F~0.2656)
[95000] TC2 MATCH for Ch0,Frame1: Expected         94 (F~0.3672), Got         94 (F~0.3672)
[95000] TC2 MATCH for Ch0,Frame2: Expected        112 (F~0.4375), Got        112 (F~0.4375)
[95000] TC2 MATCH for Ch0,Frame3: Expected         75 (F~0.2930), Got         75 (F~0.2930)
[95000] Conv2 TEST CASE 2 PASSED!
```

Figure 12: `tb_conv2_module.sv` Verification

5.7.3 Conv3

The input to this block has an input size of 4×1, with a stride size of 1, zero padding of 1, kernel size of 1, and has two kernels, which results in a 4×2 output. The figure shows the block diagram for this module.

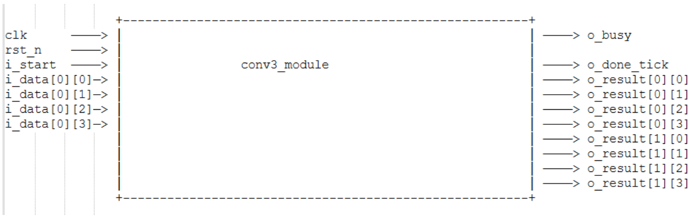
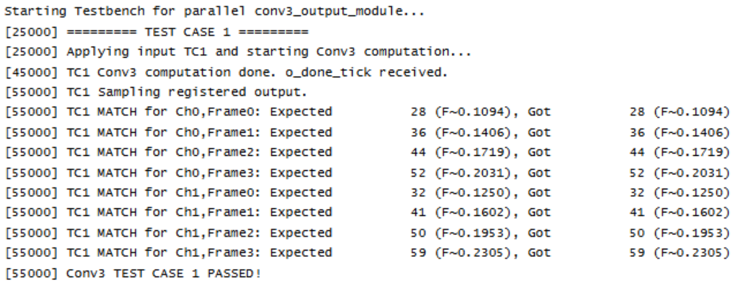


Figure 13: `conv3` module

The testbench `tb_conv3_module.sv` again checks one input against known output values. As seen below, the test passes with no resolution issues.



```
Starting Testbench for parallel conv3_output_module...
[25000] ========= TEST CASE 1 =========
[25000] Applying input TC1 and starting Conv3 computation...
[45000] TC1 Conv3 computation done. o_done_tick received.
[55000] TC1 Sampling registered output.
[55000] TC1 MATCH for Ch0,Frame0: Expected         28 (F~0.1094), Got         28 (F~0.1094)
[55000] TC1 MATCH for Ch0,Frame1: Expected         36 (F~0.1406), Got         36 (F~0.1406)
[55000] TC1 MATCH for Ch0,Frame2: Expected         44 (F~0.1719), Got         44 (F~0.1719)
[55000] TC1 MATCH for Ch0,Frame3: Expected         52 (F~0.2031), Got         52 (F~0.2031)
[55000] TC1 MATCH for Ch1,Frame0: Expected         32 (F~0.1250), Got         32 (F~0.1250)
[55000] TC1 MATCH for Ch1,Frame1: Expected         41 (F~0.1602), Got         41 (F~0.1602)
[55000] TC1 MATCH for Ch1,Frame2: Expected         50 (F~0.1953), Got         50 (F~0.1953)
[55000] TC1 MATCH for Ch1,Frame3: Expected         59 (F~0.2305), Got         59 (F~0.2305)
[55000] Conv3 TEST CASE 1 PASSED!
```

Figure 14: `tb_conv3_module.sv` Verification

HOW TO TEST: *To test these modules, go to https://www.edaplayground.com/, paste `final.sv` in `design.sv` and one of `tb_conv1_module.sv, tb_conv2_module.sv, or tb_conv2_module.sv` in `testbench.sv`, set simulator to Synopsys and run to see output.*

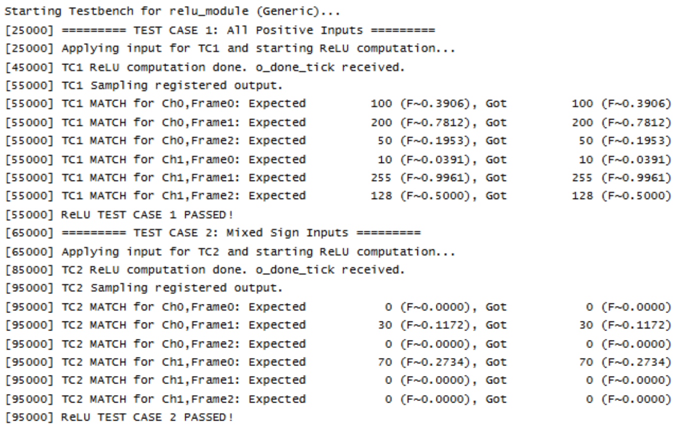
5.8 ReLU Layer

5.8.1 Design

The ReLU is very simple and is used after all convolutions to add the necessary nonlinearity. It is represented by this simple line of code: `temp_result_comb[ch][fr] = (i_data[ch][fr] < 16'sd0) ? 16'sd0 : i_data[ch][fr];`. This just takes the given value and passes it through if greater than zero and makes it zero otherwise. It has the same type of states as all other modules indicating busy or idle and takes 2 cycles to complete.

5.8.2 Verification

The testbench `tb_relu_module.sv` again checks 2 inputs, one with mixed positive and negative values and one with just positive against known output values. As seen below, the test passes with no resolution issues.



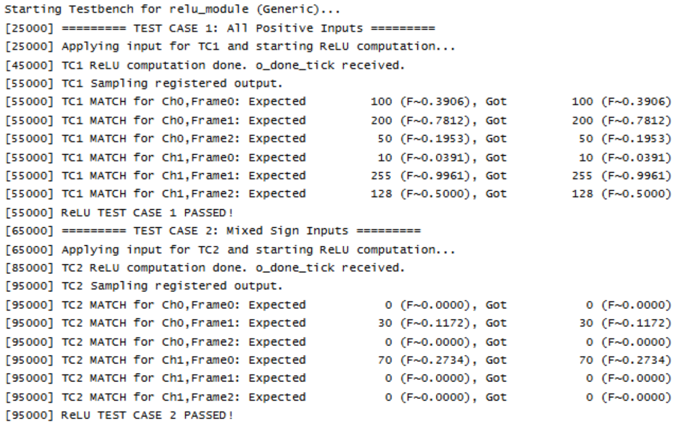
```
Starting Testbench for relu_module (Generic)...
[25000] ========= TEST CASE 1: All Positive Inputs =========
[25000] Applying input for TC1 and starting ReLU computation...
[45000] TC1 ReLU computation done. o_done_tick received.
[55000] TC1 Sampling registered output.
[55000] TC1 MATCH for Ch0,Frame0: Expected        100 (F~0.3906), Got        100 (F~0.3906)
[55000] TC1 MATCH for Ch0,Frame1: Expected        200 (F~0.7812), Got        200 (F~0.7812)
[55000] TC1 MATCH for Ch0,Frame2: Expected         50 (F~0.1953), Got         50 (F~0.1953)
[55000] TC1 MATCH for Ch1,Frame0: Expected         10 (F~0.0391), Got         10 (F~0.0391)
[55000] TC1 MATCH for Ch1,Frame1: Expected        255 (F~0.9961), Got        255 (F~0.9961)
[55000] TC1 MATCH for Ch1,Frame2: Expected        128 (F~0.5000), Got        128 (F~0.5000)
[55000] ReLU TEST CASE 1 PASSED!
[65000] ========= TEST CASE 2: Mixed Sign Inputs =========
[65000] Applying input for TC2 and starting ReLU computation...
[85000] TC2 ReLU computation done. o_done_tick received.
[95000] TC2 Sampling registered output.
[95000] TC2 MATCH for Ch0,Frame0: Expected          0 (F~0.0000), Got          0 (F~0.0000)
[95000] TC2 MATCH for Ch0,Frame1: Expected         30 (F~0.1172), Got         30 (F~0.1172)
[95000] TC2 MATCH for Ch0,Frame2: Expected          0 (F~0.0000), Got          0 (F~0.0000)
[95000] TC2 MATCH for Ch1,Frame0: Expected         70 (F~0.2734), Got         70 (F~0.2734)
[95000] TC2 MATCH for Ch1,Frame1: Expected          0 (F~0.0000), Got          0 (F~0.0000)
[95000] TC2 MATCH for Ch1,Frame2: Expected          0 (F~0.0000), Got          0 (F~0.0000)
[95000] ReLU TEST CASE 2 PASSED!
```

Figure 15: `tb_relu_module.sv` Verification

HOW TO TEST: *To test this module, go to https://www.edaplayground.com/, paste* ***`final.sv`*** *in* ***`design.sv`*** *and* ***`tb_relu_module.sv`*** *in* ***`testbench.sv`****, set simulator to Synopsys and run to see output.*

5.9 Top Level Module

The top-level module for this is a non-pipelined state-machine that checks that each stage is complete before passing the output from the completed layer to the next. The verification for this module is shown below. Note that the small mismatches at some of the layers are propagated through the network showing small errors at the output. Despite this, this is a working concept version of our decoder network and the author is very proud to have completed this! With the fully parallelized design, this should take 20 clock cycles per latent input.



```
Starting Testbench for relu_module (Generic)...
[25000] ========= TEST CASE 1: All Positive Inputs =========
[25000] Applying input for TC1 and starting ReLU computation...
[45000] TC1 ReLU computation done. o_done_tick received.
[55000] TC1 Sampling registered output.
[55000] TC1 MATCH for Ch0,Frame0: Expected        100 (F~0.3906), Got        100 (F~0.3906)
[55000] TC1 MATCH for Ch0,Frame1: Expected        200 (F~0.7812), Got        200 (F~0.7812)
[55000] TC1 MATCH for Ch0,Frame2: Expected         50 (F~0.1953), Got         50 (F~0.1953)
[55000] TC1 MATCH for Ch1,Frame0: Expected         10 (F~0.0391), Got         10 (F~0.0391)
[55000] TC1 MATCH for Ch1,Frame1: Expected        255 (F~0.9961), Got        255 (F~0.9961)
[55000] TC1 MATCH for Ch1,Frame2: Expected        128 (F~0.5000), Got        128 (F~0.5000)
[55000] ReLU TEST CASE 1 PASSED!
[65000] ========= TEST CASE 2: Mixed Sign Inputs =========
[65000] Applying input for TC2 and starting ReLU computation...
[85000] TC2 ReLU computation done. o_done_tick received.
[95000] TC2 Sampling registered output.
[95000] TC2 MATCH for Ch0,Frame0: Expected          0 (F~0.0000), Got          0 (F~0.0000)
[95000] TC2 MATCH for Ch0,Frame1: Expected         30 (F~0.1172), Got         30 (F~0.1172)
[95000] TC2 MATCH for Ch0,Frame2: Expected          0 (F~0.0000), Got          0 (F~0.0000)
[95000] TC2 MATCH for Ch1,Frame0: Expected         70 (F~0.2734), Got         70 (F~0.2734)
[95000] TC2 MATCH for Ch1,Frame1: Expected          0 (F~0.0000), Got          0 (F~0.0000)
[95000] TC2 MATCH for Ch1,Frame2: Expected          0 (F~0.0000), Got          0 (F~0.0000)
[95000] ReLU TEST CASE 2 PASSED!
```

Figure 16: `final.sv` Verification

HOW TO TEST: *To test this module, go to https://www.edaplayground.com/, paste* ***`final.sv`*** *in* ***`design.sv`*** *and* ***`tb_full_chain_final_output.sv`*** *in* ***`testbench.sv`****, set simulator to Synopsys and run to see output.*

5.10 Issues Encountered and Unfinished Work

There are several primary things of note here regarding necessary future work (other than the obvious expansion of the network to beyond toy dimensions). First of all, the SPI interface was not completed. The module for this is included in the final package, but for some reason, no LLM had any success integrating it successfully into the network. The module itself is relatively simple, but the overall network seems complex enough that an LLM could not complete this task. It would take an expert human or at least one with a lot more time (not the author right now) to integrate it. Note that the SPI transfer time SHOULD be included for the overall evaluation for the speedup of the network. This is discussed in the Final Results section.

Second, the weights are baked into the modules. This is fine for some use cases, but more flexibility would be important for loading the weights of different types of audio. A completed product in the vein of this project would need memory and an interface to load the memory to the modules.

Third are the resolution issues. These would need to be resolved or at least understood and documented before implementing in silicon.

6 OpenLane (Transistor Level Design)

6.1 Tool Issues

A quick disclaimer: getting a local version of the OpenLane design flow proved to be very difficult. After days and days spent on trying to get a working flow similar to the one shown in the CodeFest Colab example, it was decided that the Colab example could be modified to use for this design.

6.2 Final Results

The final GDS design using SkyWater 130 nm PDK provided in the Colab example, after implementation, synthesis, floorplanning, and all other steps is shown below.

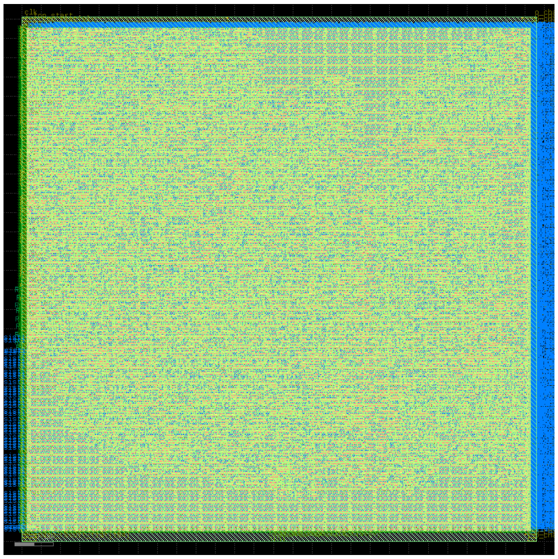
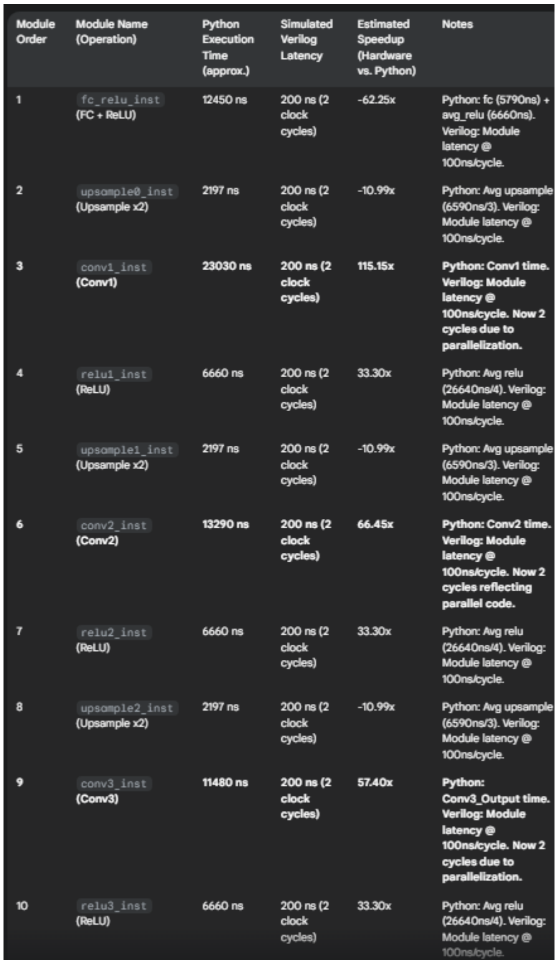


Figure 17: Final GDS design of network

This chip is 13,644 devices (transistors, resistors, etc.) and has a die size of (532 um, 542 um).

Regardless of clock speed, the worst slack hold was always in violation at the -40°C temperature corner. This was true at a clock speed of 100 MHz, 10 MHz, 1 MHz and lower. Given that this is unlikely to ever need to operate at -40°C, the lowest clock period for this is 100 ns, meaning a final total speed of 2 us per latent input. The overall speedup per layer per our benchmarking is then as follows.



Module Order	Module Name (Operation)	Python Execution Time (approx.)	Simulated Verilog Latency	Estimated Speedup (Hardware vs. Python)	Notes
1	`fc_relu_inst` (FC + ReLU)	12450 ns	200 ns (2 clock cycles)	-62.25x	Python: fc (5790ns) + avg_relu (6660ns). Verilog: Module latency @ 100ns/cycle.
2	`upsample0_inst` (Upsample x2)	2197 ns	200 ns (2 clock cycles)	-10.99x	Python: Avg upsample (6590ns/3). Verilog: Module latency @ 100ns/cycle.
3	`conv1_inst` **(Conv1)**	**23030 ns**	**200 ns (2 clock cycles)**	**115.15x**	**Python: Conv1 time. Verilog: Module latency @ 100ns/cycle. Now 2 cycles due to parallelization.**
4	`relu1_inst` (ReLU)	6660 ns	200 ns (2 clock cycles)	33.30x	Python: Avg relu (26640ns/4). Verilog: Module latency @ 100ns/cycle.
5	`upsample1_inst` (Upsample x2)	2197 ns	200 ns (2 clock cycles)	-10.99x	Python: Avg upsample (6590ns/3). Verilog: Module latency @ 100ns/cycle.
6	`conv2_inst` **(Conv2)**	**13290 ns**	**200 ns (2 clock cycles)**	**66.45x**	**Python: Conv2 time. Verilog: Module latency @ 100ns/cycle. Now 2 cycles reflecting parallel code.**
7	`relu2_inst` (ReLU)	6660 ns	200 ns (2 clock cycles)	33.30x	Python: Avg relu (26640ns/4). Verilog: Module latency @ 100ns/cycle.
8	`upsample2_inst` (Upsample x2)	2197 ns	200 ns (2 clock cycles)	-10.99x	Python: Avg upsample (6590ns/3). Verilog: Module latency @ 100ns/cycle.
9	`conv3_inst` **(Conv3)**	**11480 ns**	**200 ns (2 clock cycles)**	**57.40x**	**Python: Conv3_Output time. Verilog: Module latency @ 100ns/cycle. Now 2 cycles due to parallelization.**
10	`relu3_inst` (ReLU)	6660 ns	200 ns (2 clock cycles)	33.30x	Python: Avg relu (26640ns/4). Verilog: Module latency @ 100ns/cycle.

Figure 18: Final speedups per layer

The final speedup is then 105 us for the Python vs 2 us for the chiplet giving us around a 52.5× speedup. Note that the SPI interface was not included in the final design but was expected to take 29 clock cycles to transfer the data as designed, which would increase the total time to 4.9 us, which would still result in around a 20× speedup which is still acceptable.

HOW TO TEST: *To test the design flow, run all cells in OpenLane-Gen.ipynb in Google Colab.*

7 Conclusion

Through per-layer parallelization designed with SystemVerilog, a 20× speedup was achieved for a toy version of the decoder portion of a VAE. This serves as a proof of concept for implementing an entire decoder in hardware.