אוניברסיטת בר־אילן
Bar-Ilan University

Final Project – Transformer Accelerator

Authors:
David Poss & Dvir Fitoussi

Academic Supervisors:
Dr. Leonid Yavitz & David Freud

September 6, 2025

Contents

Abstract

Transformers have emerged as a dominant architecture in deep learning, offering unparalleled performance in tasks such as machine translation, natural language processing, and time-series prediction. However, the computational and energy demands of self-attention mechanisms present significant challenges for real-time deployment, especially on edge devices. This project addresses these challenges by designing and implementing a dedicated hardware accelerator for the self-attention operation, leveraging a Multiply–Accumulate (MAC) unit and fixed-point arithmetic to optimize efficiency. Key techniques include the adoption of Q5.10 fixed-point representation, lookup-table (LUT) based exponential approximations, piecewise linear interpolation, and truncation methods to balance accuracy and resource utilization. A Python reference model was developed to validate functionality, followed by a Verilog implementation of the self-attention module. Verification demonstrated functional correctness and a substantial reduction in runtime compared to the Python baseline. The results highlight the potential of hardware accelerators to enable energy-efficient and low-latency transformer inference on constrained platforms, paving the way for scalable deployment of advanced AI models in edge computing environments.

1 Introduction

In the modern era, machine learning and deep neural networks have become essential tools for solving complex problems across various domains, such as computer vision, natural language understanding, and predictive modeling [1, 2]. Among recent advances, the Transformer architecture has emerged as one of the most powerful models, achieving state-of-the-art results in machine translation, text analysis, image recognition, and even time-series forecasting [3, 4, 5, 6]. However, the impressive performance of large-scale Transformer models such as GPT and BERT comes at the cost of enormous computational requirements [7, 8, 9]. The requirement of high processing power places heavy strain on standard CPUs and GPUs, which are not always optimal in terms of throughput, power consumption, or cost [10, 11, 12].

To address these challenges, researchers have explored dedicated accelerators tailored to Transformer workloads, leveraging specialized hardware units such as Multiply-Accumulate (MAC) blocks and optimized memory hierarchies [13, 14, 15]. Such designs are particularly important for edge computing scenarios, where devices must deliver real-time performance under strict constraints of energy, area, and latency [16, 17, 18].

This project builds on these insights by designing and implementing a dedicated hardware accelerator for self-attention. We evaluate its efficiency through simulation, quantization strategies, and comparison with a Python reference model. The goal of this book is to detail the complete development process of the accelerator, from theoretical foundations, through design and implementation, to testing and final conclusions. Our goal is to provide readers with deep insights into the challenges of accelerating computations in deep learning and methods for addressing them using dedicated hardware.

2 Theoretical background

2.1 Transformers: A Breakthrough in Deep Learning

Transformers have revolutionized the field of deep learning by introducing an attention-based mechanism [3] that allows models to efficiently process sequential data without relying on recurrent structures. Unlike traditional recurrent neural networks (RNNs) and long short-term memory (LSTM) networks, Transformers can handle long-range dependencies more effectively by using self-attention mechanisms. The key innovation behind Transformers is the self-attention mechanism, which enables the model to weigh the importance of different words (or tokens) in a sequence relative to each other. This allows Transformers to capture contextual relationships more effectively, leading to significant improvements in tasks such as machine translation, text summarization, and speech recognition. The Transformer architecture consists of multiple layers of self-attention and feedforward neural networks. Each layer processes input sequences in parallel, making Transformers highly scalable. The computational complexity of a Transformer, however, grows quadratically with the sequence length due to the self-attention mechanism, which requires each token to attend to every other token in the sequence. This makes efficient hardware acceleration a necessity for real-time applications.

Figure 1 depicts the architecture of the Transformer model, which was introduced in the seminal paper *"Attention is All You Need"* by Vaswani et al. in 2017. This model fundamentally redefined the paradigm for sequence-to-sequence tasks by discarding recurrence and convolutions in favor of attention-based mechanisms, enabling highly parallelizable training and efficient capture of long-range dependencies. The Transformer is composed of two major components: the **encoder** on the left side and the **decoder** on the right side, each constructed by stacking N identical layers, where N is typically set to 6 in the base configuration. In the **encoder**, each layer contains two primary sub-components. First, a *multi-head self-attention mechanism* allows each position in the input sequence to attend to all other positions, capturing contextual relationships without positional bias. This self-attention mechanism operates by computing scaled dot-products between query, key, and value vectors derived from the input embeddings. To enhance the model's ability to focus on different positions and subspaces, multiple attention heads are employed in parallel, whose outputs are concatenated and linearly transformed. Following the attention block, a residual connection is added and passed through a *layer normalization* step, denoted as "Add & Norm" in the diagram. The second sub-component is a *position-wise feed-forward network* consisting of two linear transformations with a ReLU activation in between. This feed-forward block is applied independently to each position in the sequence. Again, a residual connection followed by layer normalization is applied to stabilize gradients and improve convergence. Notably, prior to entering the encoder stack, the input tokens are converted into dense vector representations via a trainable **input embedding** layer. Since the Transformer lacks recurrence and inherent positional information, *positional encodings*, often sinusoidal or learned, are added to the embeddings to enable the model to differentiate between positions in the sequence. On the **decoder** side, each layer is slightly more complex and includes three sub-components. The first is a *masked* *multi-head self-attention* mechanism. The masking is critical: it ensures that the prediction at position i can only depend on the known outputs at positions less than i, thereby preserving the autoregressive nature required for generation tasks. This is achieved by applying a mask to the attention weights, setting them to $-\infty$ for future positions before applying the softmax function. The second sub-layer is a standard *multi-head attention* mechanism, but in this case, the queries come from the decoder's previous layer, while the keys and values originate from the encoder's final output. This allows the decoder to attend over the encoded input sequence, effectively integrating the source context into the decoding process. The third sub-layer is another *feed-forward network* identical in form to that of the encoder. Each of these sub-layers is followed by residual connections and layer normalization, maintaining the "Add & Norm" structure throughout. The decoder operates on the **output embeddings**, which are obtained by shifting the target sequence one position to the right, a practice known as *teacher forcing* during training, to ensure that the prediction at each step cannot see future tokens. Positional encodings are again added to these embeddings. After passing through the N decoder layers, the output is transformed by a final *linear projection* followed by a *softmax* layer that outputs a probability distribution over the vocabulary, allowing for token-by-token generation of the output sequence. This architecture has several advantages. The use of attention allows the model to weigh the importance of different input tokens dynamically, regardless of their position, and the lack of recurrence enables significant parallelization during training. Furthermore, the modular structure of attention, normalization, and feed-forward layers provides a clean design that can be scaled or adapted to various downstream tasks. The Transformer has since become the foundation of numerous advanced language models, including BERT, GPT, T5, and many others, revolutionizing the field of natural language processing and setting new performance standards across a wide range of applications.

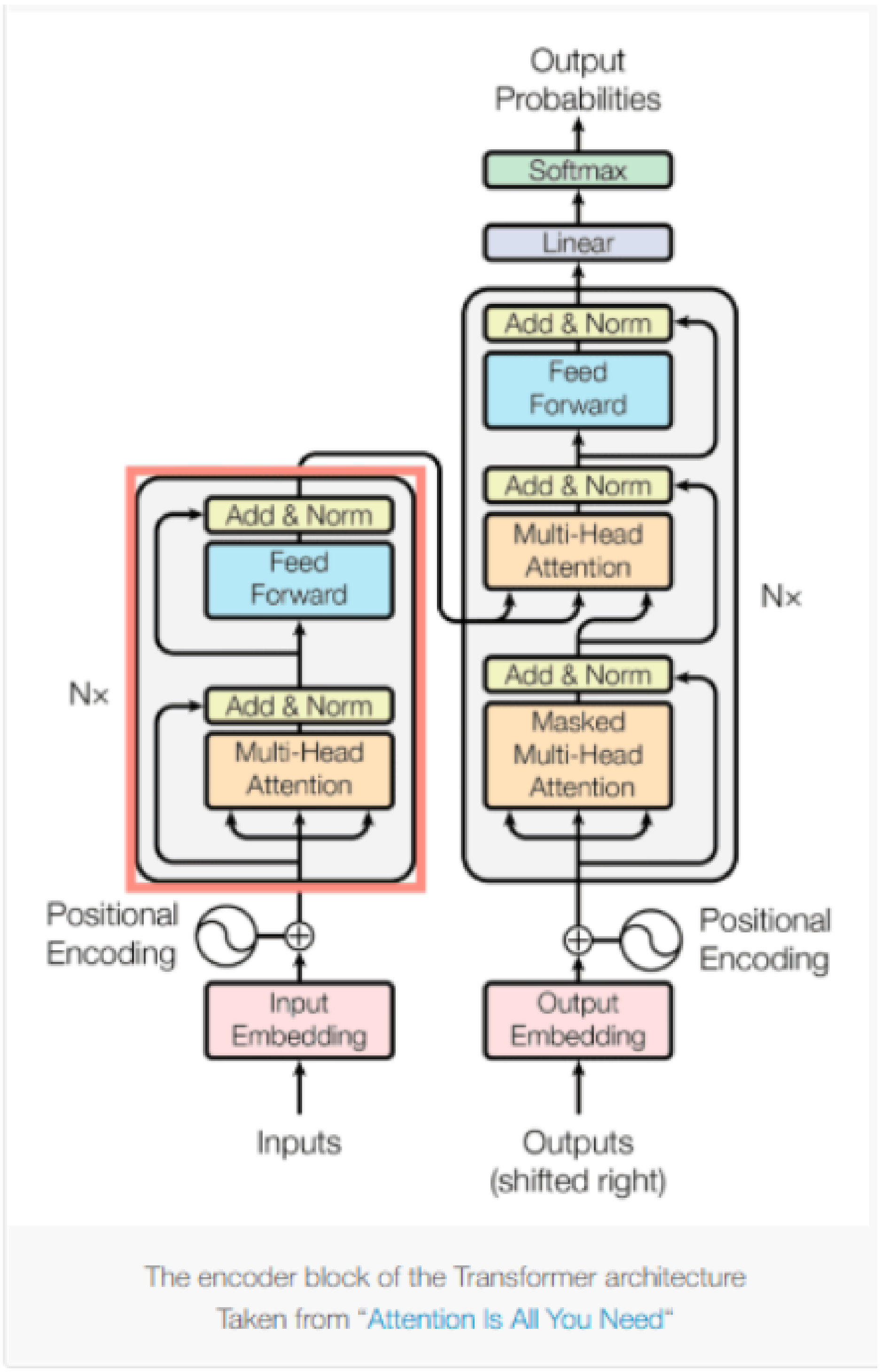

Figure 1: Self Attention encoder-decoder block architecture

2.2 Self Attention

A core innovation within the Transformer architecture is the **self-attention mechanism**, which allows the model to compute contextualized representations of each token in a sequence by attending to all other tokens, regardless of their positional distance. Traditional recurrent models process inputs sequentially and thus struggle with long-range dependencies due to vanishing gradients and limited memory. In contrast, self-attention enables direct pairwise interactions between all elements in the sequence through a parallelizable operation. Formally, as illustrated in figure 2, for an input sequence of token embeddings represented as a matrix $X \in R^{n\times d}$, where n is the sequence length and d is the embedding dimension, the model computes three learned projections: the *query* matrix $Q = XW^Q$, the *key* matrix $K = XW^K$, and the *value* matrix $V = XW^V$, where $W^Q, W^K, W^V \in R^{d\times d_k}$ are trainable parameter matrices. The attention scores are then computed via scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V.$$

The dot product $QK^\top$ measures the similarity between queries and keys, and the division by $\sqrt{d_k}$ prevents excessively large values that could saturate the softmax function, ensuring stable gradients during training. The result is a weighted sum of the value vectors V, where the weights are determined by the compatibility of each query with all keys. This operation is repeated across multiple heads in the **multi-head self-attention** mechanism, allowing the model to attend to information from different subspaces simultaneously. Each head operates independently with its own parameter matrices and outputs a representation which is then concatenated and projected back to the original dimensionality. This design enables the model to jointly capture various types of syntactic and semantic relationships within the sequence. Importantly, self-attention is permutation-invariant; thus, *positional encodings* must be added to the input embeddings to inject information about token order. Overall, self-attention is not only the foundation of the Transformer's power but also a versatile building block reused throughout the encoder, decoder, and in many subsequent architectures derived from the original Transformer model.

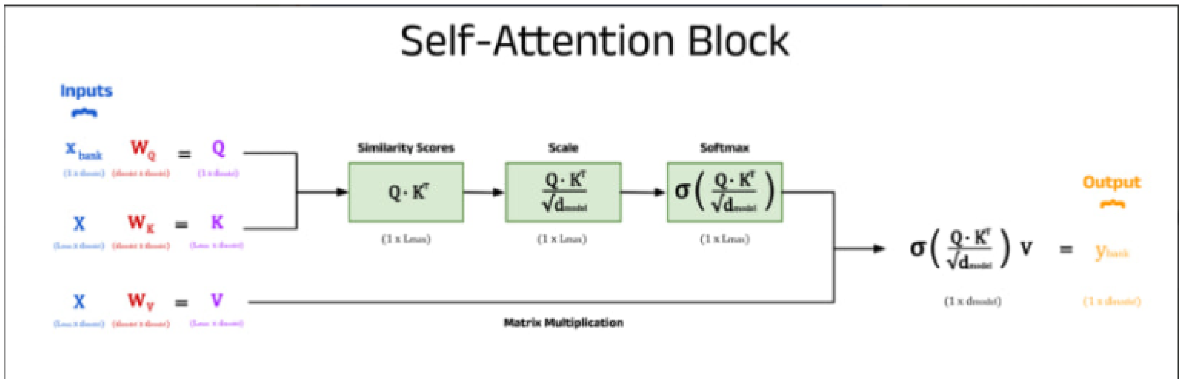

Figure 2: Self attention block

2.3 Hardware acceleration for transformers and its importance on edge devices

Transformer-based architectures have revolutionized machine learning in recent years, becoming the de facto standard in natural language processing, computer vision, time-series modeling, and beyond. However, the computational cost associated with transformers, particularly due to their multi-head attention mechanisms, large matrix multiplications, and deep sequential layers, poses a significant challenge when deploying these models in real-time applications and on resource-constrained hardware. Hardware acceleration offers a compelling solution by offloading compute-intensive operations to specialized digital logic, such as FPGAs, ASICs, or dedicated AI accelerators, thereby dramatically improving performance, energy efficiency, and scalability.

At the core of this acceleration lies the optimization of the most computationally heavy components of transformers: the matrix-vector multiplications for Query-Key-Value projections, the dot-product attention computation, and the subsequent softmax and output aggregation. By implementing these operations in custom hardware, it is possible to exploit massive parallelism, reduce memory bottlenecks, and tailor the precision to the needs of the model (e.g., fixed-point arithmetic like Q5.10) to save both area and energy.

This becomes particularly crucial in edge computing environments, where devices must operate under strict constraints of power, size, latency, and cost. Unlike cloud-based inference, edge deployment cannot rely on large-scale GPUs or access to unlimited energy sources. For example, in mobile phones, embedded medical devices, or autonomous drones, battery life is paramount and heat dissipation must be minimal. Hardware acceleration enables these devices to run transformer models with a fraction of the energy and latency of traditional CPU/GPU platforms, often reducing inference time from hundreds of milliseconds to single-digit milliseconds, while consuming milliwatts instead of watts.

Moreover, the smaller silicon footprint of well-designed accelerator modules (e.g., systolic arrays for GEMM operations, custom softmax units, quantized arithmetic blocks) allows these systems to be integrated into compact SoCs (System-on-Chip), minimizing cost and physical space. This opens the door to real-time, privacy-preserving, and always-on AI applications at the edge, ranging from on-device speech recognition and object detection to neural interfaces and sensor fusion.

In conclusion, hardware acceleration is not merely a performance optimization for transformers - it is a fundamental enabler for bringing the power of modern AI to constrained and ubiquitous computing environments. As transformer models continue to grow in capability and size, the importance of efficient hardware implementation will only increase, especially for the next generation of low-latency, low-power edge applications.

3 Methods we used

3.1 Fixed point representation

In our implementation of the Transformer architecture on hardware, we opted to use **fixed-point arithmetic** in place of floating-point operations, in order to reduce computational complexity, lower resource utilization, and ensure efficient hardware synthesis. Fixed-point representation is particularly suitable for FPGA and ASIC implementations where precision and bit-width constraints must be carefully balanced against speed and area. In fixed-point format, each number is represented using a fixed number of bits for the integer and fractional parts, respectively. To this end, we adopted the **Q5.10** format, which allocates 5 bits for the integer component, a sign bit and 10 bits for the fractional component, resulting in a total width of 16 bits. The choice of Q5.10 strikes a practical balance: it supports values in the approximate range of $[-16, +15.999]$ with a resolution of $2^{-10} \approx 0.00098$, which is sufficient to represent normalized weights, attention scores, and intermediate activations in our design with negligible quantization error. This format allowed us to keep hardware multipliers, accumulators, and memory bandwidth lean, while maintaining numerical stability across the layers of computation. Moreover, since all key operations, such as dot products, softmax approximations, and linear projections, could be efficiently implemented using integer arithmetic and bit-shifting, the Q5.10 scheme proved to be both performant and hardware-friendly. This design choice was central to achieving a lightweight and scalable self-attention module that operated entirely under a fixed 16-bit constraint without sacrificing model fidelity or functional correctness.

3.2 LUT: Look Up Table

To accelerate the computation of non-linear functions such as the exponential function e^x, we relied on **lookup tables (LUTs)**: precomputed arrays that map input indices to their corresponding function values. LUTs are particularly valuable in hardware design because they trade off memory for speed: by storing values ahead of time, we avoid costly real-time evaluations of complex mathematical operations. In our case, using a LUT allowed us to retrieve coarse approximations of e^x for integer inputs within the defined range $[-5, 4]$, with increments of 1, as depicted in table 1, eliminating the need for floating-point exponentiation units. This approach not only reduced latency and hardware resource consumption, but also formed the basis for our piecewise linear interpolation method, which refined these coarse approximations using the lower fractional bits of the input. As such, LUTs served as a foundation for a more accurate and hardware-efficient exponential approximation pipeline.

x	e^x	**Q5.10 (scaled by 1024)**
-5	$e^{-5} \approx 0.0067$	7
-4	$e^{-4} \approx 0.0183$	19
-3	$e^{-3} \approx 0.0498$	51
-2	$e^{-2} \approx 0.1353$	139
-1	$e^{-1} \approx 0.3679$	377
0	$e^0 = 1$	1024
1	$e^1 \approx 2.718$	2784
2	$e^2 \approx 7.389$	7562
3	$e^3 \approx 20.085$	20568
4	$e^4 \approx 54.598$	55809

Table 1: Lookup table for e^x used in Q5.10 exponential approximation. Values are scaled by 1024.

3.3 linear aproximation

To efficiently implement the exponential function e^x in hardware, which is a key component of the **softmax** operation in self-attention mechanisms, we employed a **piecewise linear approximation** strategy using a lookup table (LUT) and interpolation in fixed-point arithmetic. Specifically, we focused on the input domain $x \in [-5, 4]$, which covers the dynamic range relevant to most normalized dot products computed during attention score calculation. Since the exponential function is convex and smooth, it is well-suited to approximation by linear segments within bounded intervals. In our Verilog implementation, we precomputed and stored the values of e^x for integer values of $x = -5$ to 4, scaled by 2^{10} to fit the **Q5.10** fixed-point format. This 10-entry LUT was defined as a local constant array, where each entry corresponds to a scaled and signed 16-bit value representing e^x at integer x. To compute e^x

for a general input x, we extracted the integer part and the fractional part from the input fixed-point value. If the input was out of bounds (less than -5 or greater than 4), we clipped the output to e^{-5} or e^4 respectively, to maintain numerical stability.

The core idea of linear approximation is to use the first-order Taylor expansion between known data points. Given two known values $f(x_i)$ and $f(x_{i+1})$ at points x_i and x_{i+1}, we approximate $f(x)$ for $x_i \leq x \leq x_{i+1}$ using the formula

$$f(x) \approx f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}(x - x_i),$$

which becomes especially efficient in digital hardware when x_i and x_{i+1} are integers spaced by 1.

For inputs within range, we performed **linear interpolation** between two consecutive LUT values. This was done by computing the difference $\Delta = \mathrm{LUT}[i+1] - \mathrm{LUT}[i]$, then multiplying it by the fractional part of the input (i.e., the lower 10 bits), and adding the result to $\mathrm{LUT}[i]$ to yield the interpolated value. The multiplication was handled using a custom fixed-point multiply-and-truncate module to maintain precision without overflow. The final interpolated output, still in Q5.10 format, was then returned as the exponential approximation. This method allowed us to achieve both high accuracy and low resource consumption, enabling real-time computation of the softmax numerator terms using only simple arithmetic operations and a small LUT, without the need for complex transcendental hardware. Overall, this linear approximation technique was essential to making our self-attention module fully compatible with a low-bitwidth, fixed-point pipeline, while preserving functional fidelity and throughput.

Graphical Overlay

To visualize the linear approximation of the exponential function e^x, we implemented a piecewise linear method across unit-width segments of the form $[a, a+1]$, where $a \in \{-5, -4, \ldots, 3\}$, in the Desmos graphing calculator. This range was chosen to match the expected dynamic range of input values to the softmax operation, and thus corresponds directly to the domain covered by our lookup table.

The general form of each linear segment was derived from the secant line connecting (a, e^a) and $(a+1, e^{a+1})$. For any $x \in [a, a+1]$, the line is described as:

$$f(x) = \left(\frac{e^{a+1} - e^a}{(a+1) - a}\right) \cdot (x - a) + e^a = (e^{a+1} - e^a)(x - a) + e^a.$$

This expression was then gated to be active only within its domain using the indicator function $\{a \leq x \leq a+1\}$. The full approximation is the sum of these piecewise-defined lines.

$$\text{Approximation}(x) = \sum_{a=-5}^{3} \left[(e^{a+1} - e^a)(x - a) + e^a\right] \cdot \mathbf{1}_{[a,a+1]}(x),$$

where $\mathbf{1}_{[a,a+1]}(x)$ is 1 if $x \in [a, a+1]$, and 0 otherwise. Figure 3 displays the true exponential function in purple and the linear approximation in green. Each green segment corresponds to a shifted linear expression of the form above, accurately matching the exponential function at the endpoints $x = a$ and $x = a + 1$.

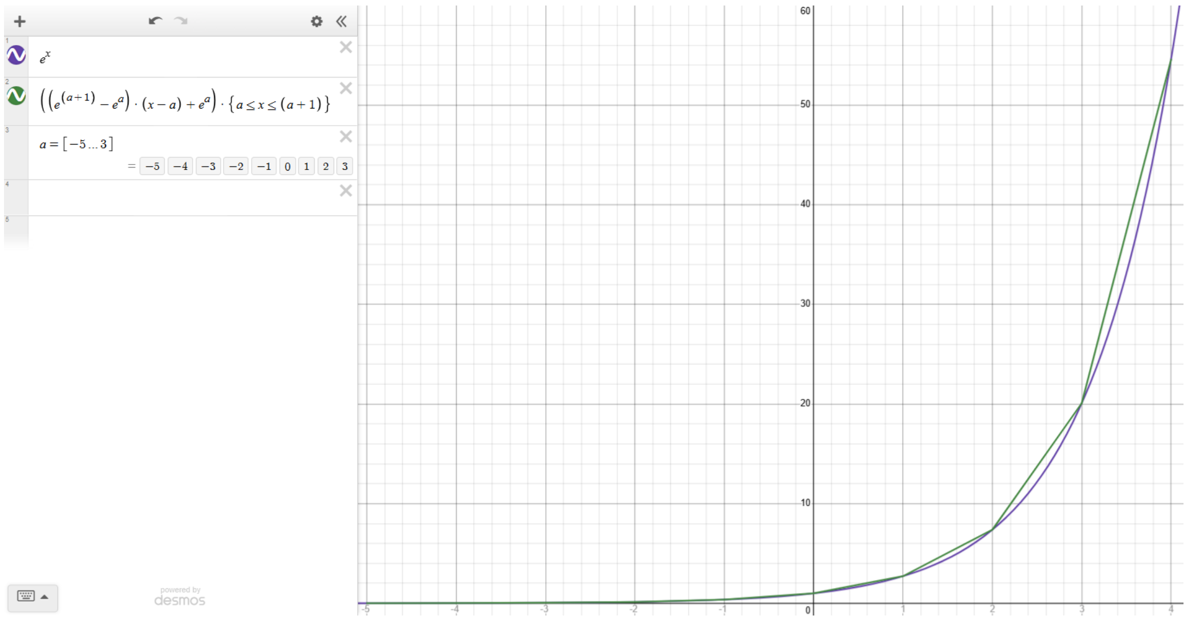

Figure 3: Overlay of the exponential function and its piecewise linear approximation for $x \in [-5, 4]$. Each segment approximates e^x between a and $a+1$, ensuring continuity and a close fit.

3.4 Truncation

In fixed-point arithmetic, multiplication of two signed numbers significantly increases the required bit-width. Specifically, when multiplying two values represented in the $\mathrm{Q}_{\mathrm{int}\cdot\mathrm{frac}}$ format, where $\mathrm{Q}_{\mathrm{int}}$ denotes the number of integer bits (including sign) and $\mathrm{Q}_{\mathrm{frac}}$ denotes the number of fractional bits, the resulting product has a width of $2 \times (Q_{\mathrm{int}} + Q_{\mathrm{frac}} + 1)$ bits to fully capture the signed result without overflow or truncation error. This expansion is particularly problematic in hardware-constrained environments such as edge devices, where power, area, and latency budgets impose strict limitations on bus widths and intermediate bit growth. To address this, we implemented a custom Verilog function `mul_and_trunc` that performs fixed-point multiplication followed by deterministic truncation, thereby limiting the output to the original bit-width ($\mathrm{Q}_{\mathrm{int}}$ + $\mathrm{Q}_{\mathrm{frac}}$ + 1) while maintaining acceptable numerical precision. The truncation method we employed involves selecting a slice of the full product that represents the result in the same $\mathrm{Q}_{\mathrm{int}\cdot\mathrm{frac}}$ format as the inputs. After computing the full-width product of the two $\mathrm{Q}_{\mathrm{int}\cdot\mathrm{frac}}$ inputs, we extract bits $[2(Q_{\mathrm{int}} + Q_{\mathrm{frac}} + 1) - 1], [2Q_{\mathrm{frac}} + Q_{\mathrm{int}} - 1 : Q_{\mathrm{frac}}]$

from the product. This corresponds to discarding the lower Q_{frac} bits, which represent the excess fractional precision, and and logically right-shifting the sign up over the extra sign and additional Q_{int} bits for modulus. Mathematically, this is equivalent to signed modulus. This process also aligns the binary point back to the original location and ensures that the output retains the same format and scaling.

Additionally, the truncation implicitly preserves the sign of the product due to the bit slicing operation starting from the original most significant bit (MSB) of the full product. Since signed fixed-point numbers in two's complement format store the sign in the MSB, keeping this bit as the new MSB during truncation is functionally equivalent to sign extension from the narrower format. Thus, the resulting truncated value continues to accurately represent negative and positive numbers without needing explicit sign extension logic. By carefully managing bit-growth in this way, we were able to maintain the integrity of fixed-point calculations within a compact 16-bit datapath, supporting efficient and scalable deep learning operations on low-power platforms. To accelerate the computation of non-linear functions such as the exponential function e^x, we relied on the aforementioned LUTs.

4 Software-to-Hardware Migration of Self-Attention

Figure 4 illustrates the high-level deployment architecture of transformer inference across cloud and edge environments. On the cloud side, transformer models are trained using high-precision floating-point arithmetic on CPU or GPU servers. Once trained, the model weights undergo a quantization step, where they are converted into a lower-precision format to optimize them for efficient execution. These quantized weights are then exported to the edge device, where a dedicated hardware accelerator, typically implemented as an ASIC, FPGA, or a custom SoC, executes the inference pipeline. This accelerator is responsible for key transformer operations, including the matrix multiplications for the Query-Key-Value projections, the attention score computation via dot products, the softmax approximation, and the final weighted aggregation of value vectors. The diagram emphasizes the contrast between traditional cloud inference (high latency and energy) and edge-based inference (low latency and low power), highlighting how hardware acceleration enables real-time, energy-efficient transformer execution on resource-constrained platforms.

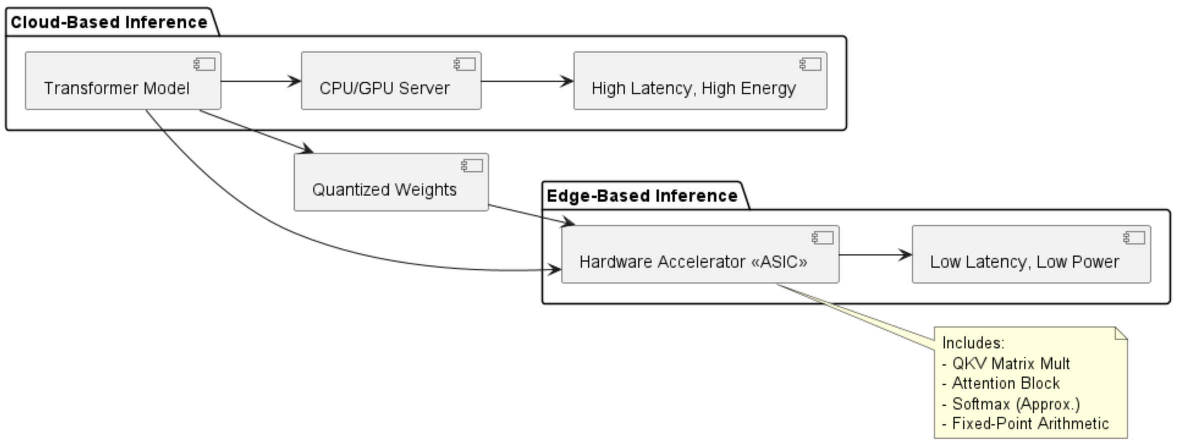

Figure 4: cloud inference vs edge-based inference

For the purpose of our project, we began by developing a self-attention reference model in Python. This model implemented the full self-attention pipeline, including the computation of query, key, and value matrices, the scaled dot product attention scores, softmax normalization, and final output aggregation. For input, we chose the sentence "*Life is short eat dessert first*" and generated deterministic embedding vectors using Python's default pseudorandom number generator with a fixed seed. These vectors, as well as the corresponding weight matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, were initially represented in standard Python floating-point (decimal) format. To migrate this setup to hardware, we converted these floating-point values to Q5.10 fixed-point representation. This involved scaling each float by $2^{10} = 1024$, rounding to the nearest integer, and expressing the result as a signed 16-bit hexadecimal literal suitable for Verilog syntax.

The Python script printed out the raw decimal values, which we then copied and translated into Verilog array literals within our testbench. These quantized arrays served as the stimuli for our Verilog self-attention module. By using the same underlying random seed and input sentence across both platforms, and by explicitly performing the Q5.10 conversion ourselves, we ensured that the Verilog module received input numerically consistent with the reference Python model. This allowed for reliable debugging and direct output comparison, validating that our fixed-point Verilog implementation produced functionally correct results in line with its floating-point Python counterpart.

5 Our code

5.1 python

```
import numpy as np
import matplotlib.pyplot as plt
import time

def softmax(x):
    x_max = np.max(x, axis=1, keepdims=True)
    e_x = np.exp(x - x_max)
    return e_x / np.sum(e_x, axis=1, keepdims=True)

def self_attention(embeddings):
    """
    Compute single-head self-attention with self-connection bias.
    Args:
        query: Input query matrix (batch_size, seq_len, d_k)
        key: Input key matrix (batch_size, seq_len, d_k)
        value: Input value matrix (batch_size, seq_len, d_k)
    Returns:
        output: Attention output
        attention_weights: Attention weight matrix
    """
    W_value, W_key, W_query, Attention_Weights, Output_Data, x =
       convert_q510_matrices_to_float()

    # Create input matrices (batch_size, seq_len, d_model)
    W_query = np.random.rand(8, 24)
    W_key = np.random.rand(8, 24)
    W_value = np.random.rand(8, 24)

    print(f"W_query shape: {W_query.shape}, W_key shape: {W_key.shape},
       W_value shape: {W_value.shape}")
    print(f"W_query: {W_query}\nW_key: {W_key}\nW_value: {W_value}\n")

    # embeddings = x
    key = np.matmul(embeddings, W_key)
    query = np.matmul(embeddings, W_query)
    value = np.matmul(embeddings, W_value)



    key = key.transpose()
    d_k = query.shape[-1]
    # print(f"query shape: {query.shape}, key shape: {key.shape},
       value shape: {value.shape}")
    scores = np.matmul(query, key) / np.sqrt(d_k)
    print(f"scores shape: {scores.shape}\n scores: {scores}")

    # Apply softmax
    #attention_weights = np.exp(scores) / np.sum(np.exp(scores), axis
       =-1, keepdims=True)
    attention_weights =softmax(scores)

    output = np.matmul(attention_weights, value)
    return output, attention_weights


def plot_attention_heatmap(attention_weights, words, title, filename):
    """
    Plot attention weights as a heatmap with word labels.
    """
    plt.figure(figsize=(6, 5))
    plt.imshow(attention_weights, cmap='viridis', interpolation='nearest
       ')
    plt.xticks(np.arange(len(words)), words, rotation=45, ha='right')
    plt.yticks(np.arange(len(words)), words)
    plt.xlabel('Key')
    plt.ylabel('Query')
    plt.title(title)
    plt.colorbar(label='Attention Weight')
    plt.savefig(filename)
    plt.close()

def convert_matrix(matrix):
    fractional_scale = 1 / 1024
    float_matrix = []
    for row in matrix:
        float_row = []
        for hex_str in row:
            int_val = int(hex_str, 16)
            if int_val > 0x7FFF:
                int_val -= 0x10000
            float_val = int_val * fractional_scale
            float_row.append(float_val)
        float_matrix.append(float_row)
    return float_matrix

def convert_q510_matrices_to_float():
    """
    Converts each matrix (W_query, W_key, W_value, Attention Weights,
       Output Data, and x)
    from Q5.10 hexadecimal to floating-point numbers

    Returns:
        W_query, W_key, W_value, Attention Weights, Output Data, and x
    """
    # Define all matrices
    w_query = [
        ["ffff", "9972", "899d", "efed", "c134", "6c05", "6b7d", "35e5",
             "63bb", "52e8", "7dd0", "f449", "68fe", "2a5b",
         "414f", "ad78", "9b0d", "5938", "efa9", "d742", "d3f5", "2161",
              "4c88", "3c1c"],
        ["5513", "fd19", "6e40", "a7f0", "778c", "816a", "ed6a", "8229",
             "28d6", "12e2", "59c6", "1c0d", "24d4", "1741",
         "9b97", "b3ac", "b4a7", "5255", "66e7", "49bf", "b41d", "c02d",
              "dc22", "a01a"],
        ["2dc7", "e7fd", "c028", "1eb9", "b5f6", "2633", "042e", "f449",
             "fde1", "400b", "7af5", "0c64", "ef16", "0a0c",
         "8a17", "2729", "669b", "3d44", "a7b1", "f069", "b669", "fdf0",
              "2cc4", "322f"],
        ["370d", "6dfa", "a20f", "19ee", "bf6e", "4b96", "3041", "86c8",
             "6c1a", "2493", "ccda", "7a6f", "3d04", "fa4a",
         "8a4e", "31fb", "fb98", "d62d", "4a54", "57ce", "3519", "4223",
              "dc31", "b674"],
        ["3111", "d964", "b3f1", "bb9a", "57f5", "42f6", "130b", "c5dd",
        "3229", "1c0d", "ba56", "4241", "7f82", "2e48",
     "b984", "0f4e", "789d", "73a4", "3572", "e60f", "01f8", "5784",
         "01c2", "a93a"],
    ["368d", "45bd", "e540", "b602", "905a", "4139", "f12e", "a2ae",
        "272c", "17e3", "6d12", "6ede", "65d2", "cebe",
     "ea68", "cef1", "e7f1", "c48b", "de59", "ce54", "370f", "ef14",
         "1e27", "19de"],
    ["c2d7", "e4ed", "e7cc", "b9a7", "a48d", "087b", "58b6", "19df",
        "9a6a", "d905", "718a", "8a1d", "f61d", "433a",
     "1bd6", "5123", "80bc", "e3f2", "8a2d", "17f0", "db66", "9186",
         "f1aa", "99d1"],
    ["b6e0", "515c", "ecc8", "da5f", "ddd1", "622c", "d01d", "6ec0",
        "5025", "ded5", "aaea", "61d3", "84ad", "3550",
     "38f9", "6b43", "82d1", "cdb3", "2b42", "8ab5", "e7f3", "0923",
         "cb41", "2cc7"],
    ["2960", "275f", "624d", "cae5", "fd59", "1e85", "71de", "07c7",
        "121f", "b0f7", "6b60", "7605", "9c18", "8c36",
     "0788", "4cb4", "e57d", "754e", "9bc0", "e0fb", "6f55", "8fa4",
         "8847", "5972"]
]

w_key = [
    ["3ffe", "2716", "d4dc", "bb8a", "3038", "ef33", "4902", "fab7",
        "739c", "c11d", "7cd3", "6d82", "705e", "9d02",
     "eb03", "8bf4", "2b95", "50bb", "22fc", "85b6", "dd7c", "36a2",
         "95ed", "18fb"],
    ["d59d", "d52d", "66a7", "cf25", "b752", "579a", "df3b", "2c80",
        "8a19", "4e9a", "05f2", "b67e", "fd22", "c314",
     "aa33", "6d06", "7f27", "5628", "b69e", "e15f", "fde6", "ba7d",
         "ca7a", "94a4"],
    ["115e", "a5bf", "9160", "2995", "58b1", "8ab1", "3e89", "051d",
        "e65a", "d110", "1089", "bd2d", "80d6", "98a0",
     "8d74", "d5ca", "8e67", "60cd", "50c4", "4e1e", "4e09", "17ce",
         "5560", "59f9"],
    ["1ede", "e7fa", "9901", "a686", "7c7b", "9d08", "f67a", "b347",
        "41b8", "43e5", "51c1", "7777", "0ebf", "d184",
     "234f", "dc6a", "ab9e", "c34f", "742f", "b193", "5631", "d147",
         "b7be", "110b"],
    ["6359", "3b62", "5443", "dee3", "127f", "3e8b", "f4f2", "e997",
        "fad3", "3621", "2939", "40d7", "bd22", "091f",
     "aeaa", "8dcc", "4cf8", "4ea5", "5553", "a36a", "deb9", "28a7",
        "8f63", "b0ca"],
    ["4eaa", "4f8c", "ea4e", "361e", "7d1b", "6ef6", "9ea9", "5b8d",
        "b0f4", "dd9d", "a897", "ef99", "ac1e", "84e9",
     "d2fb", "5316", "40da", "6994", "4377", "be2f", "285b", "1d70",
        "fd98", "6b04"],
    ["5d64", "c237", "8b85", "092a", "6b34", "d8c1", "d8b8", "0ddd",
        "c127", "eafa", "f9fd", "0b5b", "2dcf", "62c6",
     "de88", "5a1b", "5bad", "da88", "f92f", "b4b6", "1d3b", "994a",
        "5fd9", "f3d6"],
    ["48f1", "ae73", "b541", "d462", "2ed0", "f937", "b220", "7163",
        "8f0f", "fa5a", "1f13", "caae", "b00c", "860c",
     "6a0c", "2e36", "f042", "bc2e", "12f0", "858d", "c712", "4e8b",
        "48bd", "4739"],
    ["bbcf", "e3d3", "1027", "1832", "cd65", "d341", "5fe7", "d4cc",
        "7b10", "c748", "a5c7", "b468", "edac", "a6ea",
     "f1de", "07d4", "66c2", "c0e5", "de74", "7d86", "6586", "9699",
        "36e0", "2e7b"]
]

w_value = [
    ["d66c", "cf68", "8c9c", "c6de", "5e1d", "a30a", "17bd", "3d18",
        "5d19", "0d34", "c763", "cbde", "ea97", "dff2",
     "5ed5", "2d7b", "7eef", "c80f", "3a73", "0134", "5c4b", "b769",
        "0256", "af1e"],
    ["62bb", "4f73", "ae58", "3af5", "64c7", "779a", "4fa5", "4af7",
        "d5c5", "a3f2", "e324", "5880", "5aaf", "36f6",
     "0383", "ba3a", "362f", "f0fe", "6836", "190a", "a311", "3ff0",
        "f117", "038e"],
    ["ead4", "c21d", "f141", "8aec", "1a87", "a794", "6574", "d52c",
        "a766", "ce1b", "9e12", "74a0", "8b0c", "5fe0",
     "2d50", "8c4e", "49ff", "b9aa", "6207", "eeb8", "d34d", "da62",
        "59c1", "8985"],
    ["07bf", "2c99", "3d64", "41c6", "17f0", "971a", "8066", "f78c",
        "b4f7", "d1e6", "0ce6", "2375", "c66b", "83d6",
     "d50c", "9d92", "d0ec", "82bd", "59e2", "d379", "7dab", "cd49",
        "c392", "d720"],
    ["6e6f", "6fea", "ba9a", "96bc", "3956", "c8eb", "f1c5", "dec7",
        "a926", "b512", "a840", "4733", "bc60", "ac6c",
     "7d60", "5902", "ed88", "e6da", "b34c", "6fe3", "dd7d", "f77d",
         "41f5", "a2b0"],
    ["af05", "a74f", "2ef7", "701c", "7fec", "e641", "37d8", "93cf",
        "8bbc", "05c0", "3537", "6bd7", "27d4", "715d",
     "7f3e", "2a49", "29c4", "c7f7", "e121", "b27d", "c056", "8617",
         "18d0", "f295"],
    ["22d1", "88c9", "0083", "b17e", "c27e", "29e7", "1b4d", "770d",
        "e845", "2457", "4dc1", "42a4", "4efc", "d5f9",
     "7d80", "b084", "252a", "2377", "a31a", "0ec4", "96c3", "1feb",
         "db27", "b9b3"],
    ["d6af", "2780", "7555", "11f8", "4c79", "f256", "ecf4", "9241",
        "9ce7", "b356", "52d7", "ca40", "47c2", "87d9",
     "1b94", "9537", "e65d", "c4cd", "3a81", "f626", "2537", "06e8",
         "a764", "338a"],
    ["a3a8", "a74b", "a01a", "b521", "5b35", "b202", "0430", "5491",
        "6c24", "9a39", "d0fe", "9dcf", "63f8", "e843",
     "69e7", "c5c8", "b82c", "4471", "f550", "8bfe", "8c06", "b4f4",
         "f35d", "f41f"]
]

attention_weights = [
    ["00a8", "0066", "00f4", "00f4", "00f4", "0013"],
    ["005d", "00de", "0027", "00de", "00de", "00de"],
    ["00aa", "00aa", "00aa", "00aa", "00aa", "00aa"],
    ["00f6", "001f", "0005", "00f6", "00f6", "00f6"],
    ["001b", "00f3", "00f3", "00f3", "0015", "00f3"],
    ["00ed", "0013", "0143", "0010", "0068", "0143"]
]

output_data = [
    ["e6e2", "2748", "7ba6", "c127", "264a", "f0a4", "30b2", "2ae8",
        "c3f4", "489a", "1c6c", "f0b0", "d993", "43a4",
     "3b85", "30b2", "1a54", "dedc", "21d2", "0ece", "dfb1", "c388",
         "7522", "997d"],
    ["00e0", "353b", "7ff3", "d5a6", "473c", "a736", "e184", "a7ad",
        "0555", "d028", "bd34", "e6a2", "9f39", "a1bc",
     "b19e", "f8e1", "662e", "e563", "5228", "9090", "f1ef", "9951",
         "3067", "fa38"],
    ["ed2e", "e674", "3e32", "00b4", "67d0", "7c58", "db2c", "1bda",
        "69c8", "72a4", "c318", "73e2", "de20", "62a4",
     "27ac", "449e", "15f4", "2564", "39f0", "a6a6", "a814", "9f40",
         "5bf2", "4310"],
    ["0769", "0d94", "2c34", "eba7", "1625", "ef6a", "2642", "9589",
        "24a8", "5136", "d118", "7db8", "683a", "b662",
     "c87e", "1dca", "d514", "2fc8", "8603", "424b", "0890", "7b9a",
         "953d", "708b"],
    ["123b", "7060", "e231", "83b0", "331c", "b564", "6b0c", "6ca5",
        "3bf2", "5cfe", "7300", "4255", "7340", "dd6a",
     "7872", "c1ab", "6924", "5a52", "69a6", "ab11", "93ae", "a068",
         "8f95", "b374"],
    ["b978", "2a3b", "1f91", "375e", "1030", "879e", "563d", "e066",
        "7f1d", "1d09", "4567", "d926", "7a3d", "7e66",
     "176d", "3b15", "9fd4", "b765", "9410", "0942", "6dd4", "7e17",
         "9ba7", "c1c7"]
]

x = [
    ["FB98", "03FD", "0122", "F9CD", "FDAA", "06AE", "F65C", "FE4B",
        "0519"],
    ["FC87", "FD4D", "FF9E", "0609", "FD3C", "FE5F", "FE45", "08E2",
        "08D2"],
    ["0404", "018B", "02F5", "0608", "F83B", "04C0", "FAF7", "FD3C",
        "03A9"],
    ["FA48", "FFF0", "FC84", "FFE9", "F4C2", "F8D2", "FD4C", "03B7",
        "FF4D"],
    ["0003", "02C8", "FC76", "0121", "FCCB", "F8FA", "FEAD", "024E",
        "015B"],
    ["FFF4", "09B3", "01A8", "03F8", "08E7", "FACD", "FBC6", "070B",
        "FCE8"]
]


# Convert all matrices and store in a dictionary
W_query = convert_matrix(w_query)
W_key = convert_matrix(w_key)
W_value = convert_matrix(w_value)
```

```
    Attention_Weights = convert_matrix(attention_weights)
    Output_Data = convert_matrix(output_data)
    x = convert_matrix(x)
    return W_value, W_key, W_query, Attention_Weights, Output_Data, x

def main():
    # Set random seed for reproducibility
    np.random.seed(42)

    # Parameters
    batch_size = 1
    d_model = 8 # Embedding dimension
    sentence = "Life is short eat dessert first"
    words = sentence.split()
    seq_len = len(words)

    # Create fixed word embeddings (random for simplicity)
    embeddings = np.random.randn(seq_len, d_model)
    print(f"Input shape:{embeddings.shape}Input Matrix:{embeddings}\n")

    # start the performance counter
    start_time = time.perf_counter()

    # Compute self-attention
    output, attention_weights = self_attention(embeddings)

    # end the performance counter
    end_time = time.perf_counter()
    elapsed_time = end_time - start_time
    print(f"Elapsed time for self-attention computation: {elapsed_time
       :.6f} seconds")

    # Print output matrix
    print(f"\nOutput shape:{output.shape}Output Matrix:{output}")


    # Plot attention weights as heatmap
    plot_attention_heatmap(
        attention_weights,
        words,
        'Self-Attention Weights for Sentence',
        'sentence_attention_heatmap.png'
    )


if __name__ == "__main__":
    main() return 0;
}
```

5.2 Verilog

5.2.1 Self-Attention Module

```
module self_attention #(
    parameter DATA_WIDTH = 16,
    parameter Qint = 5,
    parameter Qfrac = 10,
    parameter SEQ_LEN = 6,
    parameter EMBED_DIM = 8,
    parameter DK = 24,
    parameter DV = 24,
    parameter SCALE_FACTOR = 16'h00D1
)(
    input logic [DATA_WIDTH-1:0] x [0:SEQ_LEN-1][0:EMBED_DIM-1],
    input logic [DATA_WIDTH-1:0] W_query [0:EMBED_DIM-1][0:DK-1],
    input logic [DATA_WIDTH-1:0] W_key [0:EMBED_DIM-1][0:DK-1],
    input logic [DATA_WIDTH-1:0] W_value [0:EMBED_DIM-1][0:DV-1],
    output logic [DATA_WIDTH-1:0] scores [0:SEQ_LEN-1][0:SEQ_LEN-1],
    output logic [DATA_WIDTH-1:0] attention [0:SEQ_LEN-1][0:SEQ_LEN-1],
    output logic [DATA_WIDTH-1:0] output_data [0:SEQ_LEN-1][0:DV-1]
);

    // Intermediate arrays
    logic [DATA_WIDTH-1:0] Q [0:SEQ_LEN-1][0:DK-1]; // Query matrix
    logic [DATA_WIDTH-1:0] K [0:SEQ_LEN-1][0:DK-1]; // Key matrix
    logic [DATA_WIDTH-1:0] V [0:SEQ_LEN-1][0:DV-1]; // Value matrix

    //multiply and truncate 2 Qi.f fixed point representation signed
       numbers as described in project book
    function automatic logic signed [DATA_WIDTH-1:0] mul_and_trunc (
        input logic signed [DATA_WIDTH-1:0] val1,
        input logic signed [DATA_WIDTH-1:0] val2
    );
        logic signed [2 * (Qint + Qfrac + 1) - 1:0] product;
        logic signed [(Qint + Qfrac):0] trunc;
        begin
            product = val1 * val2;
            trunc = {product[(((Qint + Qfrac + 1) * 2) - 1)], product
               [(((2 * Qfrac) + Qint) - 1):Qfrac]};
            return trunc;
        end
    endfunction

    // Exponential approximation function using LUT in Q5.10 format
        and linear interpolation (e^x is convex)
    function automatic logic signed [DATA_WIDTH-1:0] exp_approx(input
       logic signed [DATA_WIDTH-1:0] val);
        // LUT for x = -5 to 4, in Q5.10 format (scaled by 1024)
        logic [DATA_WIDTH-1:0] lut [0:9] = '{
            16'sh0007, // e^-5  0.0067 * 1024  7
            16'sh0013, // e^-4  0.0183 * 1024  19
            16'sh0033, // e^-3  0.0498 * 1024  51
            16'sh008B, // e^-2  0.1353 * 1024  139
            16'sh0178, // e^-1  0.3679 * 1024  376
            16'sh0400, // e^0 = 1.0 * 1024 = 1024
            16'sh0AE0, // e^1  2.718 * 1024  2784
            16'sh1D8A, // e^2  7.389 * 1024  7562
            16'sh5058, // e^3  20.086 * 1024  20568
            16'shDA39 // e^4  54.598 * 1024  55889
        };
        logic signed [DATA_WIDTH-1:0] int_part = {val[15:10], 10'b0}; //
             Extract integer part (Q5.10)
        logic signed [DATA_WIDTH-1:0] frac = {{6{val[15]}}, val[9:0]};
            // Fractional part (lower 10 bits)
        if (int_part < -5) begin
            return 16'h0000; // Approximate e^x  0 for x < -5
        end else if (int_part >= 4) begin
            return lut[9]; // Cap at e^4 for x > 4
        end else begin
            int index = {{16{int_part[15]}}, int_part} + 5; // Map -5 to
                 0, ..., 4 to 9
            logic signed [DATA_WIDTH-1:0] lut_val = lut[index];
            logic signed [DATA_WIDTH-1:0] lut_next = lut[index + 1];
            logic signed [DATA_WIDTH-1:0] delta = lut_next - lut_val; //
                 Difference for interpolation
            logic signed [DATA_WIDTH-1:0] increment = mul_and_trunc(delta
               , frac);
            logic signed [DATA_WIDTH-1:0] approx = lut_val + increment;
```

```
            // Interpolated value
        return approx; // Zero-extend to 32 bits
    end
endfunction

// Self-attention computation
always_comb begin
    // Declare loop variables outside the for loops to avoid
       Verilator errors
    int i, j, k;
    logic [DATA_WIDTH-1:0] temp_Q, temp_K, temp_V, temp_attention,
       sum_exp;
    logic [DATA_WIDTH-1:0] exp_vals [0:SEQ_LEN-1];

    // Step 1: Compute Q, K, V matrices
    for (i = 0; i < SEQ_LEN; i++) begin
        for (k = 0; k < DK; k++) begin
            Q[i][k] = 0;
            K[i][k] = 0;
            for (j = 0; j < EMBED_DIM; j++) begin
                temp_Q = mul_and_trunc(x[i][j], W_query[j][k]);
                Q[i][k] = Q[i][k] + temp_Q;
                temp_K = mul_and_trunc(x[i][j], W_key[j][k]);
                K[i][k] = K[i][k] + temp_K;
            end
        end
        for (k = 0; k < DV; k++) begin
            V[i][k] = 0;
            for (j = 0; j < EMBED_DIM; j++) begin
                temp_V = mul_and_trunc(x[i][j], W_value[j][k]);
                V[i][k] = V[i][k] + temp_V;
            end
        end
    end

    // Step 2: Compute scaled attention scores: Q*K'*(1/sqrt(DK))
       with additional scaling
    for (i = 0; i < SEQ_LEN; i++) begin
        for (j = 0; j < SEQ_LEN; j++) begin
```

```
                scores[i][j] = 0;
                for (k = 0; k < DK; k++) begin
                    scores[i][j] = scores[i][j] + mul_and_trunc(Q[i][k],
                        K[j][k]); //switched indices for K transpose
                end
                scores[i][j] = mul_and_trunc(scores[i][j], SCALE_FACTOR);
                    // Additional scaling by 1/sqrt(dk)
            end
        end

        // Step 3: Apply softmax to get attention weights: sigma(Q*K'/
            sqrt(DK))
        for (i = 0; i < SEQ_LEN; i++) begin
            sum_exp = 0;
            for (j = 0; j < SEQ_LEN; j++) begin
                exp_vals[j] = exp_approx(scores[i][j]);
                sum_exp = sum_exp + exp_vals[j];
            end
            for (j = 0; j < SEQ_LEN; j++) begin
                temp_attention = (exp_vals[j] << 10) / (sum_exp != 0 ?
                    sum_exp : 1);
                attention[i][j] = temp_attention[15:0]; //repair scaling
            end
        end

        // Step 4: Compute output using attention weights: sigma(QK'/
            sqrt(DK))*V
        for (i = 0; i < SEQ_LEN; i++) begin
            for (k = 0; k < DV; k++) begin
                output_data[i][k] = 0;
                for (j = 0; j < SEQ_LEN; j++) begin
                    output_data[i][k] = output_data[i][k] + mul_and_trunc
                        (attention[i][j], V[j][k]);
                end
            end
        end
    end
endmodule
```

5.2.2 Testbench Module

```
module tb_self_attention;
    // Parameters
    parameter DATA_WIDTH = 16;
    parameter SEQ_LEN = 6;
    parameter EMBED_DIM = 8; //was 16
    parameter DK = 24;
    parameter DV = 24;
    parameter SCALE_FACTOR = 16'h00D1; // Q5.10 format: 1 / sqrt(24)
        0.2041 * 1024  209

    // Signals
    logic [DATA_WIDTH-1:0] x [0:SEQ_LEN-1][0:EMBED_DIM-1];
    logic [DATA_WIDTH-1:0] W_query [0:EMBED_DIM-1][0:DK-1];
    logic [DATA_WIDTH-1:0] W_key [0:EMBED_DIM-1][0:DK-1];
    logic [DATA_WIDTH-1:0] W_value [0:EMBED_DIM-1][0:DV-1];
    logic [DATA_WIDTH-1:0] output_data [0:SEQ_LEN-1][0:DV-1];
    logic [DATA_WIDTH-1:0] attention [0:SEQ_LEN-1][0:SEQ_LEN-1];
    logic [DATA_WIDTH-1:0] scores [0:SEQ_LEN-1][0:SEQ_LEN-1];


    // Instantiate the DUT
    self_attention #(
    .DATA_WIDTH(16),
    .SEQ_LEN(6),
    .EMBED_DIM(8),
    .DK(24),
    .DV(24),
    .SCALE_FACTOR(16'h00D1)
    ) dut (
    .x(x), // input data
    .W_query(W_query),
    .W_key(W_key),
    .W_value(W_value),
    .scores(scores),
    .attention(attention),
    .output_data(output_data)
    );
```

```
task print_matrix(input logic [DATA_WIDTH-1:0] matrix [0:EMBED_DIM
    -1][0:DK-1], input string name);
$display("%s:", name);
$write("[");
for (int i = 0; i < EMBED_DIM; i++) begin
    if (i > 0) $write(",\n ");
    else $write("\n ");
    $write("[");
    for (int j = 0; j < DK; j++) begin
        $write("%04h", matrix[i][j]);
        if (j < DK-1) $write(", ");
    end
    $write("]");
end
$write("\n]");
$display("");
endtask

task print_value_matrix(input logic [DATA_WIDTH-1:0] matrix [0:
    EMBED_DIM-1][0:DV-1], input string name);
$display("%s:", name);
$write("[");
for (int i = 0; i < EMBED_DIM; i++) begin
    if (i > 0) $write(",\n ");
    else $write("\n ");
    $write("[");
    for (int j = 0; j < DV; j++) begin
        $write("%04h", matrix[i][j]);
        if (j < DV-1) $write(", ");
    end
    $write("]");
end
$write("\n]");
$display("");
endtask

task print_scores(input logic [DATA_WIDTH-1:0] matrix [0:SEQ_LEN
    -1][0:SEQ_LEN-1], input string name);
    $display("%s:", name);
    $write("[");
    for (int i = 0; i < SEQ_LEN; i++) begin
        if (i > 0) $write(",\n ");
        else $write("\n ");
        $write("[");
        for (int j = 0; j < SEQ_LEN; j++) begin
            $write("%04h", matrix[i][j]);
            if (j < SEQ_LEN-1) $write(", ");
        end
        $write("]");
    end
    $write("\n]");
    $display("");
endtask

task print_output_data(input logic [DATA_WIDTH-1:0] matrix [0:
   SEQ_LEN-1][0:DV-1], input string name);
$display("%s:", name);
$write("[");
for (int i = 0; i < SEQ_LEN; i++) begin
    if (i > 0) $write(",\n ");
    else $write("\n ");
    $write("[");
    for (int j = 0; j < DV; j++) begin
        $write("%04h", matrix[i][j]);
        if (j < DV-1) $write(", ");
    end
    $write("]");
end
$write("\n]");
$display("");
endtask

// Test stimulus
initial begin
    // Initialize input with specific 6x9 matrix in Q5.10 format
    x = '{
    '{ 16'sh01FD, 16'shFF72, 16'sh0297, 16'sh0618, 16'shFF10, 16'
   shFF10, 16'sh0651, 16'sh0312 },
'{ 16'shFE1F, 16'sh022C, 16'shFE25, 16'shFE23, 16'sh00F8, 16'
   shF859, 16'shF91A, 16'shFDC0 },
'{ 16'shFBF3, 16'sh0142, 16'shFC5E, 16'shFA5A, 16'sh05DD, 16'
   shFF19, 16'sh0045, 16'shFA4D },
'{ 16'shFDD3, 16'sh0072, 16'shFB65, 16'sh0181, 16'shFD99, 16'
   shFED5, 16'shFD98, 16'sh0769 },
'{ 16'shFFF2, 16'shFBC5, 16'sh034A, 16'shFB1E, 16'sh00D6, 16'
   shF829, 16'shFAB0, 16'sh00CA },
'{ 16'sh02F4, 16'sh00B0, 16'shFF8A, 16'shFECC, 16'shFA16, 16'
   shFD1F, 16'shFE28, 16'sh043A }
};

W_query = '{
'{ 16'sh0120, 16'sh022C, 16'sh0090, 16'sh0335, 16'sh004C, 16'
   sh03F3, 16'sh0317, 16'sh00CB, 16'sh0006, 16'sh0343, 16'sh02D4
   , 16'sh02EA, 16'sh0316, 16'sh004C, 16'sh016F, 16'sh0077, 16'
   sh0374, 16'sh027E, 16'sh0153, 16'sh0041, 16'sh013E, 16'sh014D
   , 16'sh02EB, 16'sh028D },
'{ 16'sh038C, 16'sh01E4, 16'sh007A, 16'sh02DA, 16'sh030B, 16'
   sh023F, 16'sh0316, 16'sh01FA, 16'sh0217, 16'sh01B6, 16'sh001A
   , 16'sh006E, 16'sh0020, 16'sh028C, 16'sh0142, 16'sh0209, 16'
   sh03A1, 16'sh00FF, 16'sh01A4, 16'sh0306, 16'sh00EA, 16'sh004F
   , 16'sh0129, 16'sh00A5 },
'{ 16'sh03B8, 16'sh033B, 16'sh0289, 16'sh037C, 16'sh0337, 16'
   sh00BF, 16'sh0392, 16'sh0228, 16'sh033B, 16'sh0396, 16'sh0146
   , 16'sh0071, 16'sh00E9, 16'sh01B5, 16'sh0346, 16'sh0371, 16'
   sh0007, 16'sh020B, 16'sh01AB, 16'sh00E3, 16'sh007B, 16'sh015A
   , 16'sh03C6, 16'sh014B },
'{ 16'sh0213, 16'sh02D0, 16'sh0174, 16'sh03E3, 16'sh03D9, 16'
   sh0102, 16'sh01FD, 16'sh0134, 16'sh0124, 16'sh0026, 16'sh0270
   , 16'sh0203, 16'sh0035, 16'sh011D, 16'sh03A2, 16'sh00F5, 16'
   sh0094, 16'sh01F5, 16'sh03F1, 16'sh00F8, 16'sh02B0, 16'sh030C
   , 16'sh00F3, 16'sh02EA },
'{ 16'sh0179, 16'sh0287, 16'sh0289, 16'sh0225, 16'sh005C, 16'
   sh0357, 16'sh0148, 16'sh00BF, 16'sh002A, 16'sh025D, 16'sh02B6
   , 16'sh0011, 16'sh020C, 16'sh00E8, 16'sh0295, 16'sh00B3, 16'
   sh02C3, 16'sh018C, 16'sh03BF, 16'sh008D, 16'sh015D, 16'sh0074
   , 16'sh03B3, 16'sh0382 },
'{ 16'sh0108, 16'sh02A4, 16'sh0345, 16'sh0239, 16'sh021E, 16'
   sh00F8, 16'sh005F, 16'sh0397, 16'sh039A, 16'sh0288, 16'sh015B
   , 16'sh0166, 16'sh02E7, 16'sh0397, 16'sh038C, 16'sh031F, 16'
   sh0291, 16'sh0056, 16'sh00A5, 16'sh0398, 16'sh026D, 16'sh0009
   , 16'sh0068, 16'sh02A7 },
'{ 16'sh0005, 16'sh00A5, 16'sh0232, 16'sh02C5, 16'sh029C, 16'
   sh00E6, 16'sh02D9, 16'sh00F3, 16'sh014D, 16'sh02FC, 16'sh0299
   , 16'sh0366, 16'sh02A1, 16'sh0246, 16'sh0060, 16'sh0179, 16'
   sh0110, 16'sh00FA, 16'sh03E4, 16'sh0193, 16'sh0391, 16'sh0286
   , 16'sh032E, 16'sh0203 },
'{ 16'sh024F, 16'sh01F8, 16'sh00C8, 16'sh02E4, 16'sh0120, 16'
   sh0019, 16'sh0295, 16'sh00B5, 16'sh03C3, 16'sh03D1, 16'sh03A9
   , 16'sh017B, 16'sh0010, 16'sh03B7, 16'sh01B6, 16'sh03DE, 16'
   sh03DB, 16'sh0369, 16'sh012D, 16'sh018A, 16'sh0368, 16'sh0145
   , 16'sh00AE, 16'sh023A }
};

W_key = '{
'{ 16'sh03BF, 16'sh02C9, 16'sh0248, 16'sh0064, 16'sh0276, 16'
   sh03F6, 16'sh008F, 16'sh0213, 16'sh0382, 16'sh02F7, 16'sh02CA
   , 16'sh02CF, 16'sh0170, 16'sh012D, 16'sh033D, 16'sh033E, 16'
   sh0378, 16'sh03A7, 16'sh020C, 16'sh0202, 16'sh0331, 16'sh029A
   , 16'sh02CF, 16'sh032F },
'{ 16'sh038F, 16'sh015A, 16'sh0181, 16'sh0060, 16'sh0250, 16'
   sh0025, 16'sh01DD, 16'sh022C, 16'sh0125, 16'sh025D, 16'sh001F
   , 16'sh0026, 16'sh034A, 16'sh0171, 16'sh0082, 16'sh0217, 16'
   sh0314, 16'sh00DD, 16'sh027E, 16'sh0057, 16'sh0035, 16'sh0220
   , 16'sh022A, 16'sh028D },
'{ 16'sh02E8, 16'sh03E7, 16'sh0211, 16'sh014B, 16'sh032E, 16'
   sh0115, 16'sh01C2, 16'sh0050, 16'sh001A, 16'sh03DA, 16'sh0358
   , 16'sh02C9, 16'sh01A3, 16'sh00B1, 16'sh00A0, 16'sh0100, 16'
   sh0232, 16'sh02DC, 16'sh02A4, 16'sh011F, 16'sh03D2, 16'sh02F4
   , 16'sh0238, 16'sh0272 },
'{ 16'sh01AE, 16'sh00FE, 16'sh016D, 16'sh0308, 16'sh000F, 16'
   sh0077, 16'sh002F, 16'sh002A, 16'sh036C, 16'sh02D1, 16'sh01E6
   , 16'sh0064, 16'sh01F7, 16'sh01E5, 16'sh00B1, 16'sh01BC, 16'
   sh0198, 16'sh0277, 16'sh028A, 16'sh002E, 16'sh0180, 16'sh0281
   , 16'sh0203, 16'sh036D },
'{ 16'sh02A3, 16'sh00A7, 16'sh0048, 16'sh0292, 16'sh001B, 16'
```

```
    sh0258, 16’sh03C3, 16’sh024D, 16’sh018E, 16’sh0293, 16’sh01D5
    , 16’sh022F, 16’sh03C4, 16’sh018B, 16’sh03D8, 16’sh039F, 16’
    sh00C8, 16’sh0047, 16’sh0067, 16’sh0013, 16’sh0061, 16’sh02BB
    , 16’sh0049, 16’sh0147 },
’{ 16’sh0361, 16’sh0018, 16’sh0342, 16’sh0121, 16’sh0079, 16’
    sh02C9, 16’sh0284, 16’sh0383, 16’sh02F1, 16’sh0337, 16’sh0121
    , 16’sh00B6, 16’sh0301, 16’sh033A, 16’sh03F6, 16’sh01A7, 16’
    sh017D, 16’sh031B, 16’sh015D, 16’sh03B9, 16’sh036F, 16’sh01B7
    , 16’sh0301, 16’sh0305 },
’{ 16’sh006A, 16’sh039C, 16’sh0205, 16’sh034E, 16’sh0148, 16’
    sh0395, 16’sh018F, 16’sh000B, 16’sh039F, 16’sh005D, 16’sh0147
    , 16’sh03CD, 16’sh03CD, 16’sh024B, 16’sh0287, 16’sh01CB, 16’
    sh012C, 16’sh0151, 16’sh02B1, 16’sh0302, 16’sh032B, 16’sh0329
    , 16’sh005D, 16’sh01FA },
’{ 16’sh003B, 16’sh0233, 16’sh01C4, 16’sh038D, 16’sh0167, 16’
    sh0078, 16’sh0092, 16’sh030C, 16’sh0279, 16’sh0068, 16’sh0056
    , 16’sh02CE, 16’sh004B, 16’sh034A, 16’sh02D3, 16’sh0053, 16’
    sh0057, 16’sh03F2, 16’sh017F, 16’sh017B, 16’sh0340, 16’sh03CA
    , 16’sh03F2, 16’sh0303 }
};

W_value = ’{
’{ 16’sh0181, 16’sh0056, 16’sh031C, 16’sh023C, 16’sh01B2, 16’
    sh03A0, 16’sh0072, 16’sh01F8, 16’sh000C, 16’sh01E0, 16’sh003A
    , 16’sh007A, 16’sh0078, 16’sh0299, 16’sh02FC, 16’sh0255, 16’
    sh03D9, 16’sh0180, 16’sh0125, 16’sh0379, 16’sh00E5, 16’sh03DA
    , 16’sh000C, 16’sh03E1 },
’{ 16’sh002C, 16’sh0390, 16’sh021C, 16’sh03F9, 16’sh004C, 16’
    sh0237, 16’sh03E1, 16’sh0218, 16’sh0285, 16’sh02C8, 16’sh01D1
    , 16’sh0283, 16’sh0256, 16’sh039B, 16’sh002E, 16’sh0120, 16’
    sh03CD, 16’sh0390, 16’sh01D3, 16’sh027B, 16’sh011C, 16’sh00C1
    , 16’sh01DB, 16’sh016A },
’{ 16’sh0256, 16’sh0050, 16’sh03E6, 16’sh03F2, 16’sh02CB, 16’
    sh0225, 16’sh013D, 16’sh0341, 16’sh02BD, 16’sh00A7, 16’sh03A5
    , 16’sh034A, 16’sh03CD, 16’sh02E7, 16’sh0274, 16’sh01AC, 16’
    sh03BB, 16’sh0377, 16’sh002E, 16’sh001B, 16’sh0182, 16’sh033E
    , 16’sh03F3, 16’sh009A },
’{ 16’sh0260, 16’sh0186, 16’sh03E1, 16’sh035E, 16’sh035A, 16’
    sh01E0, 16’sh01A9, 16’sh0118, 16’sh003A, 16’sh0375, 16’sh0340
```

```
   , 16'sh0400, 16'sh03FD, 16'sh0239, 16'sh0313, 16'sh03C7, 16'
   sh0366, 16'sh00FD, 16'sh01CD, 16'sh0084, 16'sh03D1, 16'sh026D
   , 16'sh00EA, 16'sh02B0 },
'{ 16'sh0279, 16'sh016F, 16'sh0074, 16'sh02B0, 16'sh0215, 16'
   sh0317, 16'sh0215, 16'sh0369, 16'sh0235, 16'sh023E, 16'sh0382
   , 16'sh019D, 16'sh0089, 16'sh001D, 16'sh0305, 16'sh027B, 16'
   sh02D1, 16'sh00DA, 16'sh008C, 16'sh000F, 16'sh0167, 16'sh025C
   , 16'sh0192, 16'sh01C0 },
'{ 16'sh039E, 16'sh0165, 16'sh020E, 16'sh0323, 16'sh0196, 16'
   sh027D, 16'sh0373, 16'sh03CC, 16'sh0097, 16'sh03B5, 16'sh01F8
   , 16'sh0108, 16'sh01D6, 16'sh03EC, 16'sh01F8, 16'sh0151, 16'
   sh0289, 16'sh00F6, 16'sh004E, 16'sh0084, 16'sh0083, 16'sh009C
   , 16'sh008E, 16'sh0290 },
'{ 16'sh00BA, 16'sh0162, 16'sh0396, 16'sh01E5, 16'sh02AC, 16'
   sh00B0, 16'sh00C5, 16'sh002A, 16'sh00AD, 16'sh011D, 16'sh00B5
   , 16'sh005B, 16'sh007B, 16'sh01D8, 16'sh00D3, 16'sh0175, 16'
   sh0203, 16'sh02C3, 16'sh0028, 16'sh0333, 16'sh0283, 16'sh0054
   , 16'sh037F, 16'sh03AF },
'{ 16'sh003F, 16'sh011C, 16'sh033A, 16'sh02FE, 16'sh00BD, 16'
   sh00D6, 16'sh017B, 16'sh01F0, 16'sh0279, 16'sh017A, 16'sh01DA
   , 16'sh02FD, 16'sh0026, 16'sh0102, 16'sh02DA, 16'sh0395, 16'
   sh020C, 16'sh0221, 16'sh006E, 16'sh01CA, 16'sh0221, 16'sh00F8
   , 16'sh0114, 16'sh0182 }
};

// Display weight matrices
$display("Initial Weight Matrices:");
print_matrix(dut.W_query, "W_query");
print_matrix(dut.W_key, "W_key");
print_value_matrix(dut.W_value, "W_value"); // Note: DV = DK, so
    size matches

#10; // Wait for computation

// Print scores (SEQ_LEN x SEQ_LEN)
print_scores(dut.scores, "scores");

print_scores(dut.attention, "attention weights");
```

```
        // Print output_data
        print_output_data(dut.output_data, "Output Data");

        $finish;
    end
endmodule


/*wider range LUT
// Exponential approximation function using LUT with interpolation in
    Q5.10 format
// LUT covers x = -5 to 4, sufficient for typical self-attention
    scores
// Q5.10 inputs can range from -32 to 31, but extreme values are
    handled as follows:
// - x < -5: returns 0 (e^x is negligible)
// - x > 4: caps at e^4 (prevents overflow in 16-bit output)
function automatic logic [31:0] exp_approx(input logic [31:0] val);
    // LUT for x = -5 to 4, in Q5.10 format (scaled by 1024)
    logic [15:0] lut [0:9] = '{
        16'h0007, // e^-5  0.0067 * 1024  7
        16'h0013, // e^-4  0.0183 * 1024  19
        16'h0033, // e^-3  0.0498 * 1024  51
        16'h008B, // e^-2  0.1353 * 1024  139
        16'h0178, // e^-1  0.3679 * 1024  376
        16'h0400, // e^0 = 1.0 * 1024 = 1024
        16'h0AE0, // e^1  2.718 * 1024  2784
        16'h1D8A, // e^2  7.389 * 1024  7562
        16'h5058, // e^3  20.086 * 1024  20568
        16'hDA39 // e^4  54.598 * 1024  55889
    };
    int int_part = $signed(val) >>> 10; // Extract integer part (Q5
        .10)
    logic [9:0] frac = val[9:0]; // Fractional part (lower 10 bits)
    if (int_part < -5) begin
        return 32'h00000000; // e^x  0 for x < -5
    end else if (int_part > 4) begin
        return {16'h0000, lut[9]}; // Cap at e^4 for x > 4
    end else begin
```

```
        int index = int_part + 5; // Map -5 to 0, ..., 4 to 9
        if (index == 9) begin
            return {16'h0000, lut[9]}; // Use e^4 directly for x = 4
        end else begin
            logic [15:0] lut_val = lut[index];
            logic [15:0] lut_next = lut[index + 1];
            logic [15:0] delta = lut_next - lut_val; // Difference for
                interpolation
            logic [25:0] product = delta * frac; // 16-bit * 10-bit =
                26-bit
            logic [15:0] increment = product >> 10; // Scale by 1/1024
            logic [15:0] approx = lut_val + increment; // Interpolated
                value
            return {16'h0000, approx}; // Zero-extend to 32 bits
        end
    end
endfunction
*/
```

6 Verification

6.1 python input and key matrices

$$Input = \begin{bmatrix} 0.4967 & -0.1383 & 0.6477 & 1.5230 & -0.2342 & -0.2341 & 1.5792 & 0.7674 \\ -0.4695 & 0.5426 & -0.4634 & -0.4657 & 0.2420 & -1.9133 & -1.7249 & -0.5623 \\ -1.0128 & 0.3142 & -0.9080 & -1.4123 & 1.4656 & -0.2258 & 0.0675 & -1.4247 \\ -0.5444 & 0.1109 & -1.1510 & 0.3757 & -0.6006 & -0.2917 & -0.6017 & 1.8523 \\ -0.0135 & -1.0577 & 0.8225 & -1.2208 & 0.2089 & -1.9597 & -1.3282 & 0.1969 \\ 0.7385 & 0.1714 & -0.1156 & -0.3011 & -1.4785 & -0.7198 & -0.4606 & 1.0571 \end{bmatrix}$$

$$W_{\text{query}} = \begin{bmatrix} 0.2809 & 0.5427 & 0.1409 & 0.8022 & 0.0746 & 0.9869 & 0.7722 & 0.1987 & 0.0055 & 0.8155 \\ 0.7069 & 0.7290 & 0.7713 & 0.0740 & 0.3585 & 0.1159 & 0.8631 & 0.6233 & 0.3309 & 0.0636 \\ 0.3110 & 0.3252 & 0.7296 & 0.6376 & & & & & & \\ 0.8872 & 0.4722 & 0.1196 & 0.7132 & 0.7608 & 0.5613 & 0.7710 & 0.4938 & 0.5227 & 0.4275 \\ 0.0254 & 0.1079 & 0.0314 & 0.6364 & 0.3144 & 0.5086 & 0.9076 & 0.2493 & 0.4104 & 0.7556 \\ 0.2288 & 0.0770 & 0.2898 & 0.1612 & & & & & & \\ 0.9297 & 0.8081 & 0.6334 & 0.8715 & 0.8037 & 0.1866 & 0.8926 & 0.5393 & 0.8074 & 0.8961 \\ 0.3180 & 0.1101 & 0.2279 & 0.4271 & 0.8180 & 0.8607 & 0.0070 & 0.5107 & 0.4174 & 0.2221 \\ 0.1199 & 0.3376 & 0.9429 & 0.3232 & & & & & & \\ 0.5188 & 0.7030 & 0.3636 & 0.9718 & 0.9624 & 0.2518 & 0.4972 & 0.3009 & 0.2848 & 0.0369 \\ 0.6096 & 0.5027 & 0.0515 & 0.2786 & 0.9083 & 0.2396 & 0.1449 & 0.4895 & 0.9857 & 0.2421 \\ 0.6721 & 0.7616 & 0.2376 & 0.7282 & & & & & & \\ 0.3678 & 0.6323 & 0.6335 & 0.5358 & 0.0903 & 0.8353 & 0.3208 & 0.1865 & 0.0408 & 0.5909 \\ 0.6776 & 0.0166 & 0.5121 & 0.2265 & 0.6452 & 0.1744 & 0.6909 & 0.3867 & 0.9367 & 0.1375 \\ 0.3411 & 0.1135 & 0.9247 & 0.8773 & & & & & & \\ 0.2579 & 0.6600 & 0.8172 & 0.5552 & 0.5297 & 0.2419 & 0.0931 & 0.8972 & 0.9004 & 0.6331 \\ 0.3390 & 0.3492 & 0.7260 & 0.8971 & 0.8871 & 0.7799 & 0.6420 & 0.0841 & 0.1616 & 0.8986 \\ 0.6064 & 0.0092 & 0.1015 & 0.6635 & & & & & & \\ 0.0051 & 0.1608 & 0.5487 & 0.6919 & 0.6520 & 0.2243 & 0.7122 & 0.2372 & 0.3254 & 0.7465 \\ 0.6496 & 0.8492 & 0.6576 & 0.5683 & 0.0937 & 0.3677 & 0.2652 & 0.2440 & 0.9730 & 0.3931 \\ 0.8920 & 0.6311 & 0.7948 & 0.5026 & & & & & & \\ 0.5769 & 0.4925 & 0.1952 & 0.7225 & 0.2808 & 0.0243 & 0.6455 & 0.1771 & 0.9405 & 0.9539 \\ 0.9149 & 0.3702 & 0.0155 & 0.9283 & 0.4282 & 0.9667 & 0.9636 & 0.8530 & 0.2944 & 0.3851 \\ 0.8511 & 0.3169 & 0.1695 & 0.5568 & & & & & & \end{bmatrix}$$

$$W_{\text{key}} = \begin{bmatrix}
0.9362 & 0.6960 & 0.5701 & 0.0972 & 0.6150 & 0.9901 & 0.1401 & 0.5183 & 0.8774 & 0.7408 \\
0.6970 & 0.7025 & 0.3595 & 0.2936 & 0.8094 & 0.8101 & 0.8671 & 0.9132 & 0.5113 & 0.5015 \\
0.7983 & 0.6500 & 0.7020 & 0.7958 & & & & & & \\
0.8900 & 0.3380 & 0.3756 & 0.0940 & 0.5783 & 0.0359 & 0.4656 & 0.5426 & 0.2865 & 0.5908 \\
0.0305 & 0.0373 & 0.8226 & 0.3602 & 0.1271 & 0.5222 & 0.7700 & 0.2158 & 0.6229 & 0.0853 \\
0.0517 & 0.5314 & 0.5406 & 0.6374 & & & & & & \\
0.7261 & 0.9759 & 0.5163 & 0.3230 & 0.7952 & 0.2708 & 0.4390 & 0.0785 & 0.0254 & 0.9626 \\
0.8360 & 0.6960 & 0.4090 & 0.1733 & 0.1564 & 0.2502 & 0.5492 & 0.7146 & 0.6602 & 0.2799 \\
0.9549 & 0.7379 & 0.5544 & 0.6117 & & & & & & \\
0.4196 & 0.2477 & 0.3560 & 0.7578 & 0.0144 & 0.1161 & 0.0460 & 0.0407 & 0.8555 & 0.7037 \\
0.4742 & 0.0978 & 0.4916 & 0.4735 & 0.1732 & 0.4339 & 0.3985 & 0.6159 & 0.6351 & 0.0453 \\
0.3746 & 0.6259 & 0.5031 & 0.8565 & & & & & & \\
0.6587 & 0.1629 & 0.0706 & 0.6424 & 0.0265 & 0.5858 & 0.9402 & 0.5755 & 0.3882 & 0.6433 \\
0.4583 & 0.5456 & 0.9415 & 0.3861 & 0.9612 & 0.9054 & 0.1958 & 0.0694 & 0.1008 & 0.0182 \\
0.0944 & 0.6830 & 0.0712 & 0.3190 & & & & & & \\
0.8449 & 0.0233 & 0.8145 & 0.2819 & 0.1182 & 0.6967 & 0.6289 & 0.8775 & 0.7351 & 0.8035 \\
0.2820 & 0.1774 & 0.7506 & 0.8068 & 0.9905 & 0.4126 & 0.3720 & 0.7764 & 0.3408 & 0.9308 \\
0.8584 & 0.4290 & 0.7509 & 0.7545 & & & & & & \\
0.1031 & 0.9026 & 0.5053 & 0.8265 & 0.3200 & 0.8955 & 0.3892 & 0.0108 & 0.9054 & 0.0913 \\
0.3193 & 0.9501 & 0.9506 & 0.5734 & 0.6318 & 0.4484 & 0.2932 & 0.3287 & 0.6725 & 0.7524 \\
0.7916 & 0.7896 & 0.0912 & 0.4944 & & & & & & \\
0.0576 & 0.5495 & 0.4415 & 0.8877 & 0.3509 & 0.1171 & 0.1430 & 0.7615 & 0.6182 & 0.1011 \\
0.0841 & 0.7010 & 0.0728 & 0.8219 & 0.7062 & 0.0813 & 0.0848 & 0.9866 & 0.3743 & 0.3706 \\
0.8128 & 0.9472 & 0.9860 & 0.7534 & & & & & &
\end{bmatrix}$$

$$W_{\text{value}} = \begin{bmatrix}
0.3763 & 0.0835 & 0.7771 & 0.5584 & 0.4242 & 0.9064 & 0.1112 & 0.4926 & 0.0114 & 0.4687 \\
0.0563 & 0.1188 & 0.1175 & 0.6492 & 0.7460 & 0.5834 & 0.9622 & 0.3749 & 0.2857 & 0.8686 \\
0.2236 & 0.9632 & 0.0122 & 0.9699 & & & & & & \\
0.0432 & 0.8911 & 0.5277 & 0.9930 & 0.0738 & 0.5539 & 0.9693 & 0.5231 & 0.6294 & 0.6957 \\
0.4545 & 0.6276 & 0.5843 & 0.9012 & 0.0454 & 0.2810 & 0.9504 & 0.8903 & 0.4557 & 0.6201 \\
0.2774 & 0.1881 & 0.4637 & 0.3534 & & & & & & \\
0.5837 & 0.0777 & 0.9744 & 0.9862 & 0.6982 & 0.5361 & 0.3095 & 0.8138 & 0.6847 & 0.1626 \\
0.9109 & 0.8225 & 0.9498 & 0.7257 & 0.6134 & 0.4182 & 0.9327 & 0.8661 & 0.0452 & 0.0264 \\
0.3765 & 0.8106 & 0.9873 & 0.1504 & & & & & & \\
0.5941 & 0.3809 & 0.9699 & 0.8421 & 0.8383 & 0.4687 & 0.4148 & 0.2734 & 0.0564 & 0.8647 \\
0.8129 & 0.9997 & 0.9966 & 0.5554 & 0.7690 & 0.9448 & 0.8496 & 0.2473 & 0.4505 & 0.1292 \\
0.9541 & 0.6062 & 0.2286 & 0.6717 & & & & & & \\
0.6181 & 0.3582 & 0.1136 & 0.6716 & 0.5203 & 0.7723 & 0.5202 & 0.8522 & 0.5519 & 0.5609 \\
0.8767 & 0.4035 & 0.1340 & 0.0288 & 0.7551 & 0.6203 & 0.7041 & 0.2130 & 0.1364 & 0.0145 \\
0.3506 & 0.5899 & 0.3922 & 0.4375 & & & & & & \\
0.9042 & 0.3483 & 0.5140 & 0.7837 & 0.3965 & 0.6221 & 0.8624 & 0.9495 & 0.1471 & 0.9266 \\
0.4921 & 0.2582 & 0.4591 & 0.9800 & 0.4926 & 0.3288 & 0.6334 & 0.2401 & 0.0759 & 0.1289 \\
0.1280 & 0.1519 & 0.1388 & 0.6409 & & & & & & \\
0.1819 & 0.3457 & 0.8968 & 0.4740 & 0.6676 & 0.1723 & 0.1923 & 0.0409 & 0.1689 & 0.2786 \\
0.1770 & 0.0887 & 0.1206 & 0.4608 & 0.2063 & 0.3643 & 0.5034 & 0.6904 & 0.0393 & 0.7994 \\
0.6279 & 0.0818 & 0.8736 & 0.9209 & & & & & & \\
0.0611 & 0.2769 & 0.8062 & 0.7483 & 0.1845 & 0.2093 & 0.3705 & 0.4845 & 0.6183 & 0.3689 \\
0.4625 & 0.7475 & 0.0367 & 0.2524 & 0.7133 & 0.8952 & 0.5117 & 0.5321 & 0.1072 & 0.4474 \\
0.5326 & 0.2425 & 0.2692 & 0.3773 & & & & & &
\end{bmatrix}$$

6.2 keys and inputs converted to fix point

$$input = \begin{bmatrix} 16'h01fd & 16'hff72 & 16'h0297 & 16'h0618 & 16'hff10 & 16'hff10 & 16'h0651 & 16'h0312 \\ 16'hfe1f & 16'h022c & 16'hfe25 & 16'hfe23 & 16'h00f8 & 16'hf859 & 16'hf91a & 16'hfdc0 \\ 16'hfbf3 & 16'h0142 & 16'hfc5e & 16'hfa5a & 16'h05dd & 16'hff19 & 16'h0045 & 16'hfa4d \\ 16'hfdd3 & 16'h0072 & 16'hfb65 & 16'h0181 & 16'hfd99 & 16'hfed5 & 16'hfd98 & 16'h0769 \\ 16'hfff2 & 16'hfbc5 & 16'h034a & 16'hfb1e & 16'h00d6 & 16'hf829 & 16'hfab0 & 16'h00ca \\ 16'h02f4 & 16'h00b0 & 16'hff8a & 16'hfecc & 16'hfa16 & 16'hfd1f & 16'hfe28 & 16'h043a \end{bmatrix}$$

$$W_{\text{query}} = \begin{bmatrix} 16'h0120 & 16'h022c & 16'h0090 & 16'h0335 & 16'h004c & 16'h03f3 & 16'h0317 & 16'h00cb & 16'h0006 & 16'h0343 \\ 16'h02d4 & 16'h02ea & 16'h0316 & 16'h004c & 16'h016f & 16'h0077 & 16'h0374 & 16'h027e & 16'h0153 & 16'h0041 \\ 16'h013e & 16'h014d & 16'h02eb & 16'h028d \\ 16'h038c & 16'h01e4 & 16'h007a & 16'h02da & 16'h030b & 16'h023f & 16'h0316 & 16'h01fa & 16'h0217 & 16'h01b6 \\ 16'h001a & 16'h006e & 16'h0020 & 16'h028c & 16'h0142 & 16'h0209 & 16'h03a1 & 16'h00ff & 16'h01a4 & 16'h0306 \\ 16'h00ea & 16'h004f & 16'h0129 & 16'h00a5 \\ 16'h03b8 & 16'h033b & 16'h0289 & 16'h037c & 16'h0337 & 16'h00bf & 16'h0392 & 16'h0228 & 16'h033b & 16'h0396 \\ 16'h0146 & 16'h0071 & 16'h00e9 & 16'h01b5 & 16'h0346 & 16'h0371 & 16'h0007 & 16'h020b & 16'h01ab & 16'h00e3 \\ 16'h007b & 16'h015a & 16'h03c6 & 16'h014b \\ 16'h0213 & 16'h02d0 & 16'h0174 & 16'h03e3 & 16'h03d9 & 16'h0102 & 16'h01fd & 16'h0134 & 16'h0124 & 16'h0026 \\ 16'h0270 & 16'h0203 & 16'h0035 & 16'h011d & 16'h03a2 & 16'h00f5 & 16'h0094 & 16'h01f5 & 16'h03f1 & 16'h00f8 \\ 16'h02b0 & 16'h030c & 16'h00f3 & 16'h02ea \\ 16'h0179 & 16'h0287 & 16'h0289 & 16'h0225 & 16'h005c & 16'h0357 & 16'h0148 & 16'h00bf & 16'h002a & 16'h025d \\ 16'h02b6 & 16'h0011 & 16'h020c & 16'h00e8 & 16'h0295 & 16'h00b3 & 16'h02c3 & 16'h018c & 16'h03bf & 16'h008d \\ 16'h015d & 16'h0074 & 16'h03b3 & 16'h0382 \\ 16'h0108 & 16'h02a4 & 16'h0345 & 16'h0239 & 16'h021e & 16'h00f8 & 16'h005f & 16'h0397 & 16'h039a & 16'h0288 \\ 16'h015b & 16'h0166 & 16'h02e7 & 16'h0397 & 16'h038c & 16'h031f & 16'h0291 & 16'h0056 & 16'h00a5 & 16'h0398 \\ 16'h026d & 16'h0009 & 16'h0068 & 16'h02a7 \\ 16'h0005 & 16'h00a5 & 16'h0232 & 16'h02c5 & 16'h029c & 16'h00e6 & 16'h02d9 & 16'h00f3 & 16'h014d & 16'h02fc \\ 16'h0299 & 16'h0366 & 16'h02a1 & 16'h0246 & 16'h0060 & 16'h0179 & 16'h0110 & 16'h00fa & 16'h03e4 & 16'h0193 \\ 16'h0391 & 16'h0286 & 16'h032e & 16'h0203 \\ 16'h024f & 16'h01f8 & 16'h00c8 & 16'h02e4 & 16'h0120 & 16'h0019 & 16'h0295 & 16'h00b5 & 16'h03c3 & 16'h03d1 \\ 16'h03a9 & 16'h017b & 16'h0010 & 16'h03b7 & 16'h01b6 & 16'h03de & 16'h03db & 16'h0369 & 16'h012d & 16'h018a \\ 16'h0368 & 16'h0145 & 16'h00ae & 16'h023a \end{bmatrix}$$

$$
\mathrm{W_{key}} = \begin{bmatrix}
16'h03bf & 16'h02c9 & 16'h0248 & 16'h0064 & 16'h0276 & 16'h03f6 & 16'h008f & 16'h0213 & 16'h0382 & 16'h02f7 \\
16'h02ca & 16'h02cf & 16'h0170 & 16'h012d & 16'h033d & 16'h033e & 16'h0378 & 16'h03a7 & 16'h020c & 16'h0202 \\
16'h0331 & 16'h029a & 16'h02cf & 16'h032f & & & & & & \\
16'h038f & 16'h015a & 16'h0181 & 16'h0060 & 16'h0250 & 16'h0025 & 16'h01dd & 16'h022c & 16'h0125 & 16'h025d \\
16'h001f & 16'h0026 & 16'h034a & 16'h0171 & 16'h0082 & 16'h0217 & 16'h0314 & 16'h00dd & 16'h027e & 16'h0057 \\
16'h0035 & 16'h0220 & 16'h022a & 16'h028d & & & & & & \\
16'h02e8 & 16'h03e7 & 16'h0211 & 16'h014b & 16'h032e & 16'h0115 & 16'h01c2 & 16'h0050 & 16'h001a & 16'h03da \\
16'h0358 & 16'h02c9 & 16'h01a3 & 16'h00b1 & 16'h00a0 & 16'h0100 & 16'h0232 & 16'h02dc & 16'h02a4 & 16'h011f \\
16'h03d2 & 16'h02f4 & 16'h0238 & 16'h0272 & & & & & & \\
16'h01ae & 16'h00fe & 16'h016d & 16'h0308 & 16'h000f & 16'h0077 & 16'h002f & 16'h002a & 16'h036c & 16'h02d1 \\
16'h01e6 & 16'h0064 & 16'h01f7 & 16'h01e5 & 16'h00b1 & 16'h01bc & 16'h0198 & 16'h0277 & 16'h028a & 16'h002e \\
16'h0180 & 16'h0281 & 16'h0203 & 16'h036d & & & & & & \\
16'h02a3 & 16'h00a7 & 16'h0048 & 16'h0292 & 16'h001b & 16'h0258 & 16'h03c3 & 16'h024d & 16'h018e & 16'h0293 \\
16'h01d5 & 16'h022f & 16'h03c4 & 16'h018b & 16'h03d8 & 16'h039f & 16'h00c8 & 16'h0047 & 16'h0067 & 16'h0013 \\
16'h0061 & 16'h02bb & 16'h0049 & 16'h0147 & & & & & & \\
16'h0361 & 16'h0018 & 16'h0342 & 16'h0121 & 16'h0079 & 16'h02c9 & 16'h0284 & 16'h0383 & 16'h02f1 & 16'h0337 \\
16'h0121 & 16'h00b6 & 16'h0301 & 16'h033a & 16'h03f6 & 16'h01a7 & 16'h017d & 16'h031b & 16'h015d & 16'h03b9 \\
16'h036f & 16'h01b7 & 16'h0301 & 16'h0305 & & & & & & \\
16'h006a & 16'h039c & 16'h0205 & 16'h034e & 16'h0148 & 16'h0395 & 16'h018f & 16'h000b & 16'h039f & 16'h005d \\
16'h0147 & 16'h03cd & 16'h03cd & 16'h024b & 16'h0287 & 16'h01cb & 16'h012c & 16'h0151 & 16'h02b1 & 16'h0302 \\
16'h032b & 16'h0329 & 16'h005d & 16'h01fa & & & & & & \\
16'h003b & 16'h0233 & 16'h01c4 & 16'h038d & 16'h0167 & 16'h0078 & 16'h0092 & 16'h030c & 16'h0279 & 16'h0068 \\
16'h0056 & 16'h02ce & 16'h004b & 16'h034a & 16'h02d3 & 16'h0053 & 16'h0057 & 16'h03f2 & 16'h017f & 16'h017b \\
16'h0340 & 16'h03ca & 16'h03f2 & 16'h0303 & & & & & &
\end{bmatrix}
$$

$$\mathrm{W}_{\mathrm{value}} = \begin{bmatrix} 16'h0181 & 16'h0056 & 16'h031c & 16'h023c & 16'h01b2 & 16'h03a0 & 16'h0072 & 16'h01f8 & 16'h000c & 16'h01e0 \\ 16'h003a & 16'h007a & 16'h0078 & 16'h0299 & 16'h02fc & 16'h0255 & 16'h03d9 & 16'h0180 & 16'h0125 & 16'h0379 \\ 16'h00e5 & 16'h03da & 16'h000c & 16'h03e1 & & & & & & \\ 16'h002c & 16'h0390 & 16'h021c & 16'h03f9 & 16'h004c & 16'h0237 & 16'h03e1 & 16'h0218 & 16'h0285 & 16'h02c8 \\ 16'h01d1 & 16'h0283 & 16'h0256 & 16'h039b & 16'h002e & 16'h0120 & 16'h03cd & 16'h0390 & 16'h01d3 & 16'h027b \\ 16'h011c & 16'h00c1 & 16'h01db & 16'h016a & & & & & & \\ 16'h0256 & 16'h0050 & 16'h03e6 & 16'h03f2 & 16'h02cb & 16'h0225 & 16'h013d & 16'h0341 & 16'h02bd & 16'h00a7 \\ 16'h03a5 & 16'h034a & 16'h03cd & 16'h02e7 & 16'h0274 & 16'h01ac & 16'h03bb & 16'h0377 & 16'h002e & 16'h001b \\ 16'h0182 & 16'h033e & 16'h03f3 & 16'h009a & & & & & & \\ 16'h0260 & 16'h0186 & 16'h03e1 & 16'h035e & 16'h035a & 16'h01e0 & 16'h01a9 & 16'h0118 & 16'h003a & 16'h0375 \\ 16'h0340 & 16'h0400 & 16'h03fd & 16'h0239 & 16'h0313 & 16'h03c7 & 16'h0366 & 16'h00fd & 16'h01cd & 16'h0084 \\ 16'h03d1 & 16'h026d & 16'h00ea & 16'h02b0 & & & & & & \\ 16'h0279 & 16'h016f & 16'h0074 & 16'h02b0 & 16'h0215 & 16'h0317 & 16'h0215 & 16'h0369 & 16'h0235 & 16'h023e \\ 16'h0382 & 16'h019d & 16'h0089 & 16'h001d & 16'h0305 & 16'h027b & 16'h02d1 & 16'h00da & 16'h008c & 16'h000f \\ 16'h0167 & 16'h025c & 16'h0192 & 16'h01c0 & & & & & & \\ 16'h039e & 16'h0165 & 16'h020e & 16'h0323 & 16'h0196 & 16'h027d & 16'h0373 & 16'h03cc & 16'h0097 & 16'h03b5 \\ 16'h01f8 & 16'h0108 & 16'h01d6 & 16'h03ec & 16'h01f8 & 16'h0151 & 16'h0289 & 16'h00f6 & 16'h004e & 16'h0084 \\ 16'h0083 & 16'h009c & 16'h008e & 16'h0290 & & & & & & \\ 16'h00ba & 16'h0162 & 16'h0396 & 16'h01e5 & 16'h02ac & 16'h00b0 & 16'h00c5 & 16'h002a & 16'h00ad & 16'h011d \\ 16'h00b5 & 16'h005b & 16'h007b & 16'h01d8 & 16'h00d3 & 16'h0175 & 16'h0203 & 16'h02c3 & 16'h0028 & 16'h0333 \\ 16'h0283 & 16'h0054 & 16'h037f & 16'h03af & & & & & & \\ 16'h003f & 16'h011c & 16'h033a & 16'h02fe & 16'h00bd & 16'h00d6 & 16'h017b & 16'h01f0 & 16'h0279 & 16'h017a \\ 16'h01da & 16'h02fd & 16'h0026 & 16'h0102 & 16'h02da & 16'h0395 & 16'h020c & 16'h0221 & 16'h006e & 16'h01ca \\ 16'h0221 & 16'h00f8 & 16'h0114 & 16'h0182 & & & & & & \end{bmatrix}$$

6.3 python results

$$\text{scores} = \begin{bmatrix} 25.6317 & -30.4785 & -20.3449 & -6.4170 & -23.2608 & -4.5623 \\ -28.3605 & 35.1110 & 22.0993 & 7.2815 & 27.7195 & 5.7290 \\ -19.2208 & 23.0530 & 15.0187 & 4.9078 & 17.6426 & 3.7762 \\ -3.5865 & 4.8337 & 2.5361 & 1.1735 & 4.1421 & 1.0965 \\ -23.4138 & 29.3212 & 18.0325 & 6.0010 & 23.0700 & 4.4766 \\ -5.5808 & 7.1805 & 4.5680 & 1.1970 & 5.8621 & 0.8327 \end{bmatrix}$$

$$\begin{matrix}\text{Attention}\\ \text{Weights}\end{matrix} = \begin{bmatrix} 1.00000 & 4.28191\times10^{-25} & 1.07799\times10^{-20} & 1.20617\times10^{-14} & 5.83808\times10^{-22} & 7.70725\times10^{-14} \\ 2.71921\times10^{-28} & 9.99381\times10^{-1} & 2.23271\times10^{-6} & 8.19531\times10^{-13} & 6.16132\times10^{-4} & 1.73506\times10^{-13} \\ 4.35172\times10^{-19} & 9.95228\times10^{-1} & 3.22618\times10^{-4} & 1.31088\times10^{-8} & 4.44880\times10^{-3} & 4.22787\times10^{-9} \\ 1.33476\times10^{-4} & 6.05690\times10^{-1} & 6.08679\times10^{-2} & 1.55830\times10^{-2} & 3.03297\times10^{-1} & 1.44271\times10^{-2} \\ 1.24923\times10^{-23} & 9.98063\times10^{-1} & 1.24892\times10^{-5} & 7.43549\times10^{-11} & 1.92429\times10^{-3} & 1.61901\times10^{-11} \\ 2.13327\times10^{-6} & 7.43399\times10^{-1} & 5.45258\times10^{-2} & 1.87332\times10^{-3} & 1.98897\times10^{-1} & 1.30140\times10^{-3} \end{bmatrix}$$

$$\text{Output} = \begin{bmatrix} 1.4415 & 1.1417 & 4.3094 & 3.0434 & 2.9107 & 1.5410 & 1.0177 & 1.1304 & 1.0256 & 1.9337 \\ 2.1072 & 2.5864 & 2.1905 & 2.1990 & 2.5139 & 3.0008 & 3.1192 & 2.3933 & 0.8890 & 2.1317 \\ 3.0579 & 2.0421 & 2.3914 & 3.0446 & & & & & & \\ -2.6285 & -1.1015 & -3.9373 & -3.1478 & -2.7607 & -2.0100 & -1.9277 & -2.4050 & -0.7941 & -2.6463 \\ -1.8753 & -1.5319 & -1.7174 & -3.2159 & -2.4842 & -2.3656 & -2.9616 & -2.1073 & -0.3584 & -2.0173 \\ -2.1159 & -1.4328 & -2.1465 & -3.5669 & & & & & & \\ -2.6263 & -1.1060 & -3.9336 & -3.1476 & -2.7577 & -2.0096 & -1.9316 & -2.4025 & -0.7927 & -2.6499 \\ -1.8742 & -1.5325 & -1.7190 & -3.2168 & -2.4801 & -2.3640 & -2.9614 & -2.1059 & -0.3616 & -2.0171 \\ -2.1157 & -1.4295 & -2.1427 & -3.5658 & & & & & & \\ -2.3595 & -1.4085 & -3.5265 & -3.0380 & -2.4376 & -1.9205 & -2.1443 & -2.1223 & -0.6589 & -2.8220 \\ -1.7503 & -1.5684 & -1.8276 & -3.1765 & -2.0965 & -2.1748 & -2.8697 & -1.9270 & -0.6125 & -1.9178 \\ -2.0344 & -1.1717 & -1.7669 & -3.3275 & & & & & & \\ -2.6279 & -1.1031 & -3.9361 & -3.1478 & -2.7598 & -2.0100 & -1.9291 & -2.4043 & -0.7936 & -2.6476 \\ -1.8750 & -1.5320 & -1.7179 & -3.2163 & -2.4828 & -2.3650 & -2.9616 & -2.1069 & -0.3595 & -2.0173 \\ -2.1159 & -1.4316 & -2.1453 & -3.5667 & & & & & & \\ -2.4504 & -1.3060 & -3.7315 & -3.1089 & -2.5609 & -1.9553 & -2.0731 & -2.2233 & -0.7131 & -2.7792 \\ -1.7998 & -1.5990 & -1.8106 & -3.2197 & -2.2639 & -2.2868 & -2.9294 & -2.0130 & -0.5378 & -1.9828 \\ -2.0950 & -1.2818 & -1.8967 & -3.4349 & & & & & & \end{bmatrix}$$

6.4 python graph output

Figure 5 illustrates the attention weight connections between tokens in the sentence under analysis. Each node represents a word, while the directed edges indicate the strength of attention from one token to another, with edge thickness and associated numerical values corresponding to the magnitude of the attention weight. Self-loops (highlighted in red) denote the proportion of attention a word allocates to itself. For example, “Life” exhibits a relatively

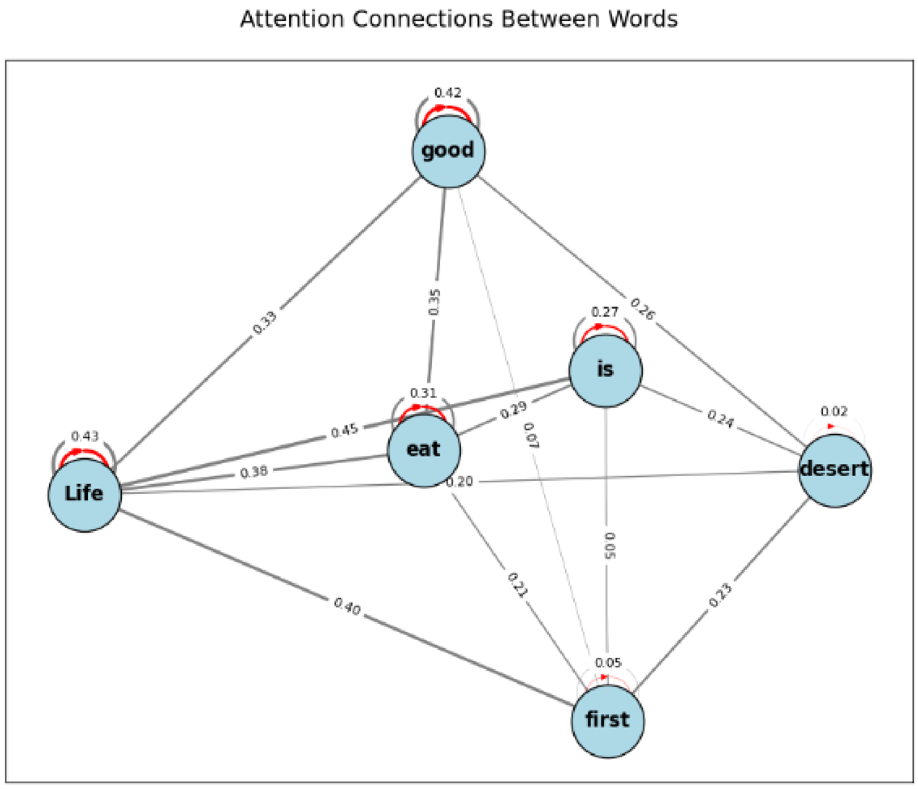

Figure 5: attention between words

high self-attention score of 0.43 and also directs notable attention toward “eat” (0.45) and “good” (0.33). In contrast, “desert” has minimal self-attention (0.02) and receives weaker attention overall. This visualization provides an interpretable representation of the learned attention distribution, enabling qualitative inspection of how contextual relationships are modeled by the self-attention mechanism

6.5 verilog results

$$\text{scores} = \begin{bmatrix} \texttt{16'hfdab} & \texttt{16'hee9f} & \texttt{16'h1712} & \texttt{16'he631} & \texttt{16'h0b6f} & \texttt{16'hed9a} \\ \texttt{16'hf718} & \texttt{16'hf014} & \texttt{16'hf036} & \texttt{16'he919} & \texttt{16'h06be} & \texttt{16'h1720} \\ \texttt{16'he752} & \texttt{16'hf409} & \texttt{16'h0815} & \texttt{16'h13cd} & \texttt{16'h129c} & \texttt{16'h0f42} \\ \texttt{16'hf183} & \texttt{16'h138d} & \texttt{16'h0a4c} & \texttt{16'h04c0} & \texttt{16'h10bf} & \texttt{16'h046d} \\ \texttt{16'h0adc} & \texttt{16'h0d21} & \texttt{16'h1426} & \texttt{16'h1833} & \texttt{16'hf41c} & \texttt{16'h1217} \\ \texttt{16'he98d} & \texttt{16'he8b3} & \texttt{16'h126c} & \texttt{16'h04da} & \texttt{16'h17a4} & \texttt{16'h0363} \end{bmatrix}$$

$$\text{attention weights} = \begin{bmatrix} \texttt{16'h0000} & \texttt{16'h0000} & \texttt{16'h04d6} & \texttt{16'h0000} & \texttt{16'h04d6} & \texttt{16'h0000} \\ \texttt{16'h0000} & \texttt{16'h0000} & \texttt{16'h0000} & \texttt{16'h0000} & \texttt{16'h04d6} & \texttt{16'h04d6} \\ \texttt{16'h0000} & \texttt{16'h0000} & \texttt{16'h0852} & \texttt{16'h0852} & \texttt{16'h0852} & \texttt{16'h0852} \\ \texttt{16'h0000} & \texttt{16'h0d01} & \texttt{16'h0d01} & \texttt{16'h0d01} & \texttt{16'h0d01} & \texttt{16'h0d01} \\ \texttt{16'h0d01} & \texttt{16'h0d01} & \texttt{16'h0d01} & \texttt{16'h0d01} & \texttt{16'h0000} & \texttt{16'h0d01} \\ \texttt{16'h0000} & \texttt{16'h0000} & \texttt{16'h05b9} & \texttt{16'h05b9} & \texttt{16'h05b9} & \texttt{16'h0041} \end{bmatrix}$$

$$\text{Output Data} = \begin{bmatrix} \texttt{16'hf01c} & \texttt{16'hf2ea} & \texttt{16'hdeaf} & \texttt{16'he44f} & \texttt{16'hed7d} & \texttt{16'hf0b0} & \texttt{16'hee3d} & \texttt{16'hf18f} & \texttt{16'hfad8} & \texttt{16'he6e8} \\ \texttt{16'hf168} & \texttt{16'heb9d} & \texttt{16'heb78} & \texttt{16'he3ae} & \texttt{16'hed7c} & \texttt{16'he9db} & \texttt{16'he574} & \texttt{16'hef35} & \texttt{16'hf694} & \texttt{16'heeff} \\ \texttt{16'hec0f} & \texttt{16'hf3ac} & \texttt{16'hf5b2} & \texttt{16'he630} & & & & & & \\ \texttt{16'hee03} & \texttt{16'hf1df} & \texttt{16'hf265} & \texttt{16'hecfe} & \texttt{16'hf05a} & \texttt{16'hf222} & \texttt{16'hec69} & \texttt{16'hf144} & \texttt{16'hfc37} & \texttt{16'he930} \\ \texttt{16'hf0dd} & \texttt{16'hf717} & \texttt{16'hf231} & \texttt{16'hed6c} & \texttt{16'hf65c} & \texttt{16'hf57b} & \texttt{16'hee67} & \texttt{16'hf76e} & \texttt{16'hfa56} & \texttt{16'hf984} \\ \texttt{16'hf3d0} & \texttt{16'hfbff} & \texttt{16'hf591} & \texttt{16'hecc0} & & & & & & \\ \texttt{16'hcd1a} & \texttt{16'he598} & \texttt{16'hc4c2} & \texttt{16'hc55f} & \texttt{16'hcc51} & \texttt{16'hd3f1} & \texttt{16'hd7d6} & \texttt{16'hd557} & \texttt{16'hf40b} & \texttt{16'hcaf3} \\ \texttt{16'hd536} & \texttt{16'hdea6} & \texttt{16'hce6d} & \texttt{16'hc45b} & \texttt{16'hd9a4} & \texttt{16'hdc96} & \texttt{16'hc32f} & \texttt{16'hddd8} & \texttt{16'hf051} & \texttt{16'he7bf} \\ \texttt{16'hdb74} & \texttt{16'hde37} & \texttt{16'hdbe0} & \texttt{16'hc8c9} & & & & & & \\ \texttt{16'h8e37} & \texttt{16'hc85b} & \texttt{16'h702c} & \texttt{16'h7b63} & \texttt{16'h8b48} & \texttt{16'ha0fb} & \texttt{16'ha824} & \texttt{16'h9e08} & \texttt{16'he2ec} & \texttt{16'h8aa4} \\ \texttt{16'ha4af} & \texttt{16'hb7ec} & \texttt{16'h9c23} & \texttt{16'h78e6} & \texttt{16'ha3b4} & \texttt{16'ha9d6} & \texttt{16'h7a65} & \texttt{16'haf28} & \texttt{16'he2cc} & \texttt{16'hbfd0} \\ \texttt{16'hab50} & \texttt{16'hb87b} & \texttt{16'hab92} & \texttt{16'h7b46} & & & & & & \\ \texttt{16'hbd24} & \texttt{16'hf5db} & \texttt{16'hcec8} & \texttt{16'hcbea} & \texttt{16'hcbd2} & \texttt{16'hcea8} & \texttt{16'hdcf5} & \texttt{16'hc4fe} & \texttt{16'hf613} & \texttt{16'hd365} \\ \texttt{16'hd57d} & \texttt{16'hee67} & \texttt{16'hd3a8} & \texttt{16'hc351} & \texttt{16'hd6f8} & \texttt{16'hea15} & \texttt{16'hc955} & \texttt{16'he558} & \texttt{16'hfd50} & \texttt{16'hf5e2} \\ \texttt{16'hedd1} & \texttt{16'hd9de} & \texttt{16'hdafa} & \texttt{16'hcf09} & & & & & & \\ \texttt{16'he574} & \texttt{16'hf0e5} & \texttt{16'hd665} & \texttt{16'hdbe6} & \texttt{16'he2e6} & \texttt{16'he692} & \texttt{16'he9dc} & \texttt{16'he917} & \texttt{16'hf995} & \texttt{16'he144} \\ \texttt{16'heaa5} & \texttt{16'hea59} & \texttt{16'he21f} & \texttt{16'hd8b0} & \texttt{16'he8ab} & \texttt{16'he8ea} & \texttt{16'hd9e6} & \texttt{16'he877} & \texttt{16'hf54b} & \texttt{16'heb89} \\ \texttt{16'he95f} & \texttt{16'hea66} & \texttt{16'hec0e} & \texttt{16'hdd31} & & & & & & \end{bmatrix}$$

6.6 performance

To establish a performance baseline for subsequent hardware implementation, the self-attention computation was first executed in a Python environment. The measured elapsed time for a single forward pass was 0.004143 seconds. This value serves as the software benchmark against which the Verilog-based accelerator will be compared, enabling a direct evaluation of the speedup achieved through hardware optimization.

6.7 Cycle Count Estimation for Self-Attention

To estimate the hardware cost of the self-attention computation, we assume a single multiply–accumulate (MAC) unit that performs one operation per clock cycle. For the given

Table 2: Comparison of runtime between Python implementation and hardware estimation.

Implementation	**Cycles**	**Time @ 100 MHz**	**Time @ 200 MHz**
Python (CPU)	—	4.143 ms	4.143 ms
Hardware (Verilog, 1 MAC)	5184	51.8 μs	25.9 μs

configuration with input shape $(6, 8)$, weight matrices W_q, W_k, W_v of shape $(8, 24)$, and output dimension $(6, 24)$, the required cycle counts are as follows:

- **Linear projections (Q, K, V):** Each projection involves $6 \times 24 \times 8 = 1152$ MACs. Since there are three such projections, the total is

$$3 \times 1152 = 3456 \text{ cycles.}$$

- **Score matrix (QK^T):** Computing $Q(6 \times 24) \cdot K^T(24 \times 6)$ requires

$$6 \times 6 \times 24 = 864 \text{ cycles.}$$

- **Output (softmax$(QK^T)V$):** The multiplication $(6 \times 6) \cdot (6 \times 24)$ costs

$$6 \times 24 \times 6 = 864 \text{ cycles.}$$

The total cycle count is therefore

$$3456 + 864 + 864 = 5184 \text{ cycles.}$$

Assuming a clock frequency of 100 MHz (period of 10 ns), this corresponds to a latency of approximately 51.8 μs. At 200 MHz, the latency reduces to 25.9 μs.

6.8 Runtime Comparison: Python vs. Hardware

Table 2 summarizes the runtime results. The Python implementation required 0.004143 s to complete the self-attention operation, while the hardware design, assuming a single MAC unit, completes the same computation in 5184 cycles. This translates to approximately 51.8 μs at 100 MHz or 25.9 μs at 200 MHz. Thus, the hardware achieves a speedup of roughly two orders of magnitude compared to the Python baseline.

7 Results discussion

During our testing and validation process, we observed noticeable numerical discrepancies between the outputs of our self-attention block implemented in Verilog and its floating-point reference in Python. The input to both systems was identical: the sentence "*Life is short eat dessert first*" was used to generate embeddings and projection matrices in Python, which were printed to the screen and then copied into the Verilog testbench after being converted to Q5.10 fixed-point format. This ensured an apples-to-apples comparison at the input level.

The most significant source of output mismatch stemmed from our hardware approximation of the exponential function within the softmax operation. While Python used precise floating-point exponentiation, our Verilog implementation employed a compact lookup table for integer values between -4 and 5, using linear interpolation to estimate intermediate results. This method is efficient and well-suited to hardware, but it sacrifices precision, especially in the positive range of the exponential, where e^x grows rapidly and small interpolation errors become magnified. Because softmax is a normalized function where each output depends on the ratio of exponentiated terms, these errors propagate nonlinearly. Even minor differences in one or two terms can shift the entire probability distribution, particularly when the denominator is dominated by a single large value.

A secondary but important contributor to the discrepancy was the behavior of fixed-point multiplication and truncation in Verilog. Multiplying two Q5.10 values produces a 32-bit intermediate result. To return to Q5.10 format, we must truncate this product by discarding lower bits, but doing so requires precise alignment of the new integer and fractional boundaries. We found no widely accepted standard for truncation in Q5.10 arithmetic, and our implementation relied on custom bit slicing. Without proper handling of sign extension and scaling, this truncation step can introduce bias or magnitude distortion, especially when high bits are shifted left or right without considering the position of the original sign bit. These effects accumulate through chained operations such as matrix multiplications and softmax normalization.

Overall, while both exponential approximation and truncation contribute to the divergence, our findings suggest that the approximation of e^x within the softmax block has the largest impact on output alignment between Verilog and Python. This underscores the sensitivity of attention mechanisms to numerical precision, and the need for careful treatment of nonlinear operations when transitioning from high-level floating-point models to resource-constrained fixed-point hardware. This can be considered one of the tradeoff considerations in compact hardware planning.

8 Potential improvements and further discussion

8.1 Limitations of Floating-Point Support in Verilator

In our effort to simulate a hardware implementation of the attention mechanism, we considered using publicly available floating-point arithmetic packages written in SystemVerilog. Several such libraries exist on GitHub and are designed to offer IEEE-754 compliant behavior suitable for hardware synthesis and simulation.

However, these implementations often rely on features introduced in modern versions of SystemVerilog (e.g., SystemVerilog-2017 and later), such as advanced `typedef`s, interface constructs, enhanced package modularity, and explicit floating-point encoding primitives. Unfortunately, our simulation flow was constrained by the use of Verilator, which, despite being a fast and powerful open-source simulator, **does not fully support modern SystemVerilog standards or IEEE-754 floating-point arithmetic**.

This limitation ruled out the possibility of directly simulating floating-point behavior in hardware, which would have brought the hardware results in closer numerical agreement with our Python reference implementation. While alternatives such as rewriting these libraries for compatibility, or invoking DPI-C modules, exist in principle, they were deemed too resource-intensive or brittle to justify within our development scope.

Instead, we adopted a fixed-point Q5.10 representation, converting floating-point values from Python into fixed-point format for input and output. These fixed-point matrices were printed and manually copied from the Python side into the Verilog testbench. While this provided a consistent data representation across systems, some differences in precision and normalization are expected, especially in operations such as exponentiation and softmax computation, which are highly sensitive to dynamic range. Proper implementation may yield a more fascinating comparison.

Following are the helper functions we implemented in the attempt to use the floating point module libraries available online. These may be used in future improvement attempts:

```
/*
==========================================================================

Floating point addition handler
==========================================================================


*/
    // Declare signals (example widths for IEEE 754 double)
    logic [63:0] adder_input_a, adder_input_b, adder_output_z;
    logic adder_input_a_stb, adder_input_b_stb;
    logic adder_output_z_ack, adder_output_z_stb;
    logic adder_input_a_ack, adder_input_b_ack;
    logic clk, adder_rst;
```

```
// Instantiate 32-bit floating point adder
double_adder u_double_adder (
    .input_a (adder_input_a),
    .input_b (adder_input_b),
    .input_a_stb (adder_input_a_stb),
    .input_b_stb (adder_input_b_stb),
    .output_z_ack (adder_output_z_ack),
    .clk (clk),
    .rst (adder_rst),
    .output_z (adder_output_z),
    .output_z_stb (adder_output_z_stb),
    .input_a_ack (adder_input_a_ack),
    .input_b_ack (adder_input_b_ack)
);

task automatic perform_addition(
input logic [63:0] a,
input logic [63:0] b,
output logic [63:0] result
);
    // Drive inputs
    adder_input_a = a;
    adder_input_b = b;
    adder_input_a_stb = 1;
    adder_input_b_stb = 1;

    // Wait until adder accepts the inputs
    wait (adder_input_a_ack && adder_input_b_ack);
    @(posedge clk);
    adder_input_a_stb = 0;
    adder_input_b_stb = 0;

    // Wait for result to be ready
    wait (adder_output_z_stb);
    result = adder_output_z;

    // Acknowledge result
    adder_output_z_ack = 1;
```

```
        @(posedge clk);
        adder_output_z_ack = 0;
    endtask


/*
==========================================================================

Floating point multiplication handler
==========================================================================

*/
    // Declare signals (example widths for IEEE 754 single-precision)
    logic [63:0] multiplier_input_a, multiplier_input_b,
       multiplier_output_z;
    logic multiplier_input_a_stb, multiplier_input_b_stb;
    logic multiplier_output_z_ack, multiplier_output_z_stb;
    logic multiplier_input_a_ack, multiplier_input_b_ack;
    logic multiplier_rst;

    // Instantiate 32-bit floating point multiplier
    double_multiplier u_double_multiplier (
        .input_a (multiplier_input_a),
        .input_b (multiplier_input_b),
        .input_a_stb (multiplier_input_a_stb),
        .input_b_stb (multiplier_input_b_stb),
        .output_z_ack (multiplier_output_z_ack),
        .clk (clk),
        .rst (multiplier_rst),
        .output_z (multiplier_output_z),
        .output_z_stb (multiplier_output_z_stb),
        .input_a_ack (multiplier_input_a_ack),
        .input_b_ack (multiplier_input_b_ack)
    );

    task automatic perform_multiplication(
        input logic [63:0] a,
        input logic [63:0] b,
        output logic [63:0] result
);
    // Drive inputs
    multiplier_input_a = a;
    multiplier_input_b = b;
    multiplier_input_a_stb = 1;
    multiplier_input_b_stb = 1;

    // Wait until multiplier accepts the inputs
    wait (multiplier_input_a_ack && multiplier_input_b_ack);
    @(posedge clk);
    multiplier_input_a_stb = 0;
    multiplier_input_b_stb = 0;

    // Wait for result to be ready
    wait (multiplier_output_z_stb);
    result = multiplier_output_z;

    // Acknowledge result
    multiplier_output_z_ack = 1;
    @(posedge clk);
    multiplier_output_z_ack = 0;
endtask
```

8.2 Architectural Trade-Offs in Matrix Computation

Beyond numeric representation, our implementation also highlights fundamental space-time trade-offs in the hardware realization of matrix-based attention mechanisms.

One option to accelerate computation without significantly increasing area is to use a systolic array. In such architectures, rows and columns of input matrices are fed into a grid of multiply-accumulate (MAC) units in a pipelined fashion. This enables partial results to propagate through the array, achieving throughput of one result per cycle after an initial latency. For a 24×24 matrix, a systolic array of $24^2 = 576$ MACs could, in principle, perform the entire matrix multiplication in approximately $O(n)$ time.

This approach is particularly compelling for repeated or batched inference on edge devices with modest FPGA/ASIC capabilities, as it enables **parallel execution** with good data locality. However, it comes at the cost of silicon area, routing complexity, and control overhead. Still, this could be considered a sweet-spot in spacial-temporal considerations. and would be interesting to test in further development.

At the opposite extreme, one may achieve maximal space efficiency by instantiating only a single MAC unit, and performing the entire computation sequentially in $O(n^2)$ time. While this drastically reduces the number of required logic elements, it eliminates any potential for real-time or high-throughput inference. However, this approach is relatively uninteresting from a hardware design perspective, since any CPU or micro-controller with sufficient memory could emulate such behavior in software.

Similarly, we may improve maximal time efficiency by using a separate MAC per calculation, thus achieving constant time. This, however, is extremely spatially inefficient, so it doesn't help for edge devices.

When approaching the design with strategies such as systolic arrays, it is important to bear in mind the scalability of the solution (although edge devices typically don't require much versatility). a systolic array, for example, can't handle ever-growing matrices without the help of remedial complex logic controllers.

9 Conclusion and Outlook

In summary, our current implementation balances these trade-offs by selecting a fixed-point format easily compatible with Verilator, and accepting the slight deviations from the Python floating-point model. While this introduces challenges in accurately modeling operations like softmax, it keeps the design compatible with open-source tools and low-cost workflows.

Future extensions could explore migrating to a more fully-featured SystemVerilog simulator with IEEE-754 support, or using external floating-point co-processors. Additionally, implementing a partial or full systolic array architecture would improve execution speed while remaining within reasonable area bounds, offering a meaningful path forward in deploying neural attention on constrained hardware platforms.

References

[1] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444.

[2] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning.* (Vol. 1, No. 2). Cambridge: MIT press.

[3] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30.

[4] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of NAACL-HLT*, 4171–4186.

[5] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... & Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations (ICLR).*

[6] Li, S., Jin, X., Xuan, Y., Zhou, X., Chen, W., Wang, Y. X., & Yan, X. (2019). Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *Advances in Neural Information Processing Systems (NeurIPS)*, 32.

[7] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361.*

[8] Narayanan, D., Shoeybi, M., Casper, J., Le, M., Mishra, S., Micikevicius, P., ... & Zaharia, M. (2021). Efficient large-scale language model training on GPU clusters. *arXiv preprint arXiv:2104.04473.*

[9] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053.*

[10] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ... & Laudon, J. (2017). In-datacenter performance analysis of a tensor processing unit. *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 1–12.

[11] Han, S., Mao, H., & Dally, W. J. (2016). Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *International Conference on Learning Representations (ICLR).*

[12] Sze, V., Chen, Y. H., Yang, T. J., & Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12), 2295–2329.

[13] Li, H., Gong, L., Zeng, Z., Zhang, X., & Li, Z. (2022). FTRANS: Energy-efficient FPGA accelerator for transformer-based language models. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(3), 1–26.

[14] Ham, T. J., Lee, Y., Seo, S. H., Kim, S., Choi, H., Jung, S. J., & Lee, J. W. (2021). ELSA: Hardware–software co-design for efficient, lightweight self-attention mechanism in neural networks. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (pp. 692–705). IEEE.

[15] Laguna, A. F., Sharifi, M. M., Kazemi, A., Yin, X., Niemier, M., & Hu, X. S. (2022). Hardware–software co-design of an in-memory transformer network accelerator. *Frontiers in Electronics*, 3, 847069. https://doi.org/10.3389/felec.2022.847069

[16] Lane, N. D., Bhattacharya, S., Mathur, A., Forlivesi, C., Kawsar, F., Seneviratne, A., ... & Zhao, F. (2016). DeepX: A software accelerator for low-power deep learning inference on mobile devices. *Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 1–12.

[17] Ji, Y., Fang, C., Ma, S., Shao, H., Wang, Z. (2024). Co-Designing Binarized Transformer and Hardware Accelerator for Efficient End-to-End Edge Deployment. *arXiv*.

[18] Banbury, C., Reddi, V. J., Torelli, P., Holleman, J., Jeffries, N., Kiraly, C., ... & Whatmough, P. (2021). MLPerf Tiny benchmark. *Proceedings of the 38th International Conference on Machine Learning (ICML)*, 129–141.