Final Project Report First Page. Must match this format (Title)

Name: Gunavant Setty Unityid: gsetty StudentID: 200597215

Delay (ns to run provided provided example). Clock period: 7.12 ns # "cycles": 3384 cycles	Logic: 9026 cells Area: 13790.7701 (um^2) Memory: N/A	1/(delay.area): 3.00955-9 $ns^{-1}.um^{-2}$
Delay (TA provided example. TA to complete)		1/(delay.area) (TA)

Abstract

This project implements a hardware module for the Scaled Dot-Product Attention mechanism, a key component of transformer models used in modern natural language processing (NLP). The design efficiently realizes the complete self-attention computation, including parallel matrix multiplications for Query (Q), Key (K), and Value (V) matrices, followed by attention score calculation and final output generation. The hardware architecture leverages both result and scratchpad SRAMs for optimal data management. The Q, K, V, and S matrices are stored concurrently in the scratchpad SRAM and result SRAM, enabling efficient reuse for subsequent computations such as the score (S) and final output (Z) matrices. This approach maximizes data locality and minimizes memory access latency, contributing to the overall performance of the attention mechanism. The purpose of storing them is to utilize them when needed for calculation of matrices like S and Z.

VLSI Implementation of Transformer Scaled Dot-Product Attention Module for NLP

GUNAVANT SETTY

ABSTRACT

This project implements a hardware module for the Scaled Dot-Product Attention mechanism, a key component of transformer models used in modern natural language processing (NLP). The design efficiently realizes the complete self-attention computation, including parallel matrix multiplications for Query (Q), Key (K), and Value (V) matrices, followed by attention score calculation and final output generation. The hardware architecture leverages both result and scratchpad SRAMs for optimal data management. The Q, K, V, and S matrices are stored concurrently in the scratchpad SRAM and result SRAM, enabling efficient reuse for subsequent computations such as the score (S) and final output (Z) matrices. This approach maximizes data locality and minimizes memory access latency, contributing to the overall performance of the attention mechanism. The purpose of storing them is to utilize them when needed for calculation of matrices like S and Z.

INTRODUCTION

This project implements a hardware accelerator for the Scaled Dot-Product Attention mechanism, a critical component of transformer models used in modern natural language processing and AI applications. The design efficiently realizes the complete self-attention computation, including parallel matrix multiplications for Query (Q), Key (K), and Value (V) matrices, followed by attention score calculation and final output generation.

Transformer models have revolutionized the field of natural language processing (NLP) and artificial intelligence (AI), offering significant improvements in processing sequential data for large language models, machine translation, and sentiment analysis. Introduced by Vaswani et al. in their seminal paper "Attention is All You Need," transformers overcome limitations of previous architectures like Long Short-Term Memory (LSTM) networks and Convolutional Neural Networks (CNNs) by processing input sequences in parallel rather than sequentially.

The transformer architecture employs several key mechanisms:

1. **Positional Encoding and Embedding**: This process maps words to vector representations of size $dmodel$, preserving contextual similarity while reducing dimensionality. Positional encoding is applied to retain sequence order information.

2. **Self-Attention**: Also known as intra-attention, this mechanism relates different positions within a single sequence to compute its representation. It allows the model to focus on relevant parts of the input when processing each element. The “Scaled Dot-Product Attention” is calculated by the formula:

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

   where Q, K, and V represent the Query, Key, and Value matrices respectively.

3. **Multi-Head Attention (MHA)**: This component allows the model to attend to information from different representation simultaneously, enhancing its ability to capture complex relationships within the data. Performs attention function with multiple $headi$, each with different, learned linear projections parameter matrices W_i^Q, K_i^K, and V_i^V.

$$MultiHead(Q, K, V) = Concat(head_1, head_2, \dots, head_h)W^O$$
$$where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

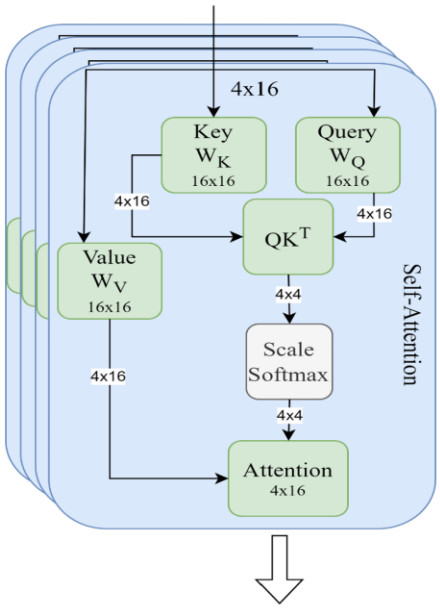


Figure 1: Transformer self-attention query, key, value, score and attention connection and matrix dimensions.

The figure above illustrates architecture of the Scaled Dot-Product Attention mechanism. The key components depicted in the diagram are:

1. **Query (Q), Key (K), and Value (V) Matrices**:
   - The input sequence is transformed into three matrices: Query (Q), Key (K), and Value (V), each through a linear transformation using learned weight matrices *WQ*, *WK*, and *WV* respectively. In this case, each matrix has dimensions of 4x16.

2. **Matrix Multiplication (Q . *KT*)**:
   - The Query matrix is multiplied by the transpose of the Key matrix *KT*, resulting in a 4x4 score matrix. This multiplication helps capture relationships between different positions in the sequence.

3. **Scaled Softmax**:
   - The score matrix can scaled by dividing by $\sqrt{dk}$ (where *dk* is the dimension of the Key vectors, in this case, 16). But in this project, this is not implemented. This scaling ensures more stable gradients during training.
   - The scaled scores are then passed through a softmax function to normalize them, turning them into attention weights.

4. **Attention Output**:
   - The attention weights are used to compute a weighted sum of the Value matrix (V). This produces the final attention output, which has dimensions 4x16.

Key innovations in this implementation include:

1. **Efficient memory management**: The architecture leverages both result and scratchpad SRAMs for optimal data storage and retrieval. Q, K, V, and S matrices are stored concurrently in the scratchpad SRAM and result SRAM, enabling efficient reuse for subsequent computations.
2. **Parallel processing**: The design employs matrix multiplications to access input and weight matrices from their respective SRAMs simultaneously, enhancing overall performance.

Results achieved:

1. Successful implementation of the complete Scaled Dot-Product Attention mechanism in hardware.
2. Efficient data management and reuse through strategic use of SRAM resources.
3. Significant speedup compared to software implementations, making it suitable for integration into larger transformer-based AI systems.

The rest of this report is structured as follows:

1. Micro-Architecture
2. Interface Specification
3. Technical Implementation
4. Verification
5. Results Achieved
6. Conclusions
7. Future Work
8. Symbol Explanation

MICRO-ARCHITECTURE

Hardware Algorithmic Approach

The hardware implementation of the **Scaled Dot-Product Attention** mechanism follows the algorithmic steps used in transformer models, specifically focusing on matrix multiplications and softmax scaling. The approach involves the following key steps:

1. **Matrix Multiplication for Query (Q), Key (K), and Value (V):**
    - The input sequence is transformed into Query (Q), Key (K), and Value (V) matrices by multiplying the input embeddings with weight matrices *WQ*, *WK*, and *WV* respectively. These matrices are stored in SRAM for efficient access during computation.
2. **Dot-Product Attention Calculation:**
    - The Query matrix is multiplied by the transpose of the Key matrix (*KT*) to compute a score matrix. This operation captures relationships between different positions in the input sequence.
3. **Attention Output:**
    - The attention weights are used to compute a weighted sum of the Value matrix, producing the final attention output, which is then used for further processing in the transformer model.

High-Level Architecture Drawing and Data Flow

The provided diagram illustrates the high-level architecture of the Scaled Dot-Product Attention mechanism:Self-Attention Architecture

- **Input Dimensions:** The input sequence has dimensions of m1xn1.
- **Query, Key, Value Matrices:** The input embeddings are multiplied by weight matrices *WQ*, *WK*, and *WV* (each with dimensions m2xn2) to produce Query, Key, and Value matrices of size m1xn2.
- **Dot Product:** The Query matrix is multiplied by the transpose of the Key matrix (*KT*) to produce a score matrix of size m1xm1.
- **Attention Output:** The attention weights are used to compute a weighted sum of the Value matrix, resulting in an output matrix of size m1xn2.

Claimed Innovations

1. **Efficient Memory Management:**
    - The design uses both result SRAM and scratchpad SRAM to store intermediate matrices (Q, K, V, S). This allows for efficient reuse of data during computations, reducing memory access latency.
2. **Parallel Matrix Multiplications:**
    - The hardware accelerates performance by performing parallel matrix multiplications for Q, K, and V matrices simultaneously. This significantly reduces computation time compared to sequential processing.
3. **Optimized Data Flow:**
    - By storing Q, K, V, and S matrices in both result SRAM and scratchpad SRAM concurrently, the design minimizes memory bottlenecks and ensures

faster access during subsequent operations like calculating attention scores or final outputs.

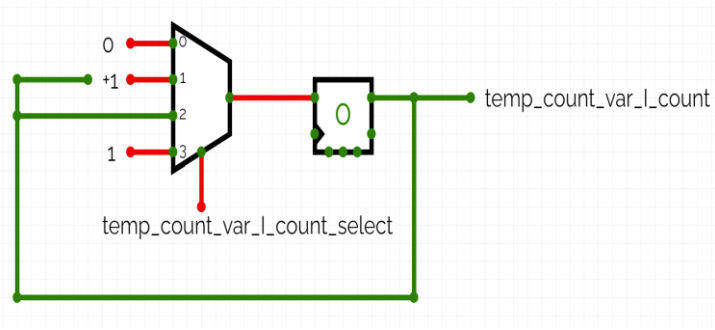
Data Flow Description

1. Input embeddings are loaded into SRAM.
2. Q, K, and V matrices are computed in parallel using respective weight matrices.
3. Q is multiplied by *KT* to compute the score matrix.
4. Attention weights are applied to the Value matrix to produce the final attention output.

This micro-architecture efficiently implements the Scaled Dot-Product Attention mechanism with optimized memory usage and parallel computation techniques for improved performance in transformer-based models.

Data path:

1. This register temp_count_var_I variable is to track I, that is if calculation of a C is done then start from 1 because in address 0 the dimensions of the matrix are stored



2. Another multiplexer for tracking of the performs matrix multiplication and accumulation, specifically for calculating the result of the equation Z=A×B+C. The code handles the accumulation of partial products and writing the results to memory. Below is an explanation of the data path depending on the matrices to read.

   Case 1: When Q_K_read == 1'b0:
   This case handles operations when reading from Q and K matrices has not started yet and data is read from input and weight SRAMs.
   - If temp_count_z_select == 2'b00, and if it is at the start of a new row (indicated by temp_count_var_I_count == 2'b0) and writing is enabled (write_enable_sel == 1'b1), then reset accum_result to zero.
   - For other values of temp_count_z_select, similar logic applies, but instead of resetting, it accumulates results using mac_result_z.

   Case 2: When Q_K_read == 1'b1 && S_V_read == 1'b0:
   This case handles operations when reading from Q and K matrices has finished, but reading from V has not started yet.

- Similar logic applies here as in Case 1, where different actions are taken based on the value of temp_count_z_select.
- If at the start of a new row (temp_count_var_I_count == 32'b0) and writing is enabled (write_enable_sel == 1'b1), then reset accum_result.

Case 3: When Q_K_read == 1'b1 && S_V_read == 1'b1:
This case handles operations when both Q-K and S-V reads are complete, meaning we are now working on accumulating results for Z.

- The same logic applies here as in previous cases, but now it uses another counter (counter_for_V) to track progress through matrix V.
- If at the start of a new row (counter_for_V == 1) and writing is enabled (write_enable_sel == 2'b1), then reset accum_result.

Final Accumulation Logic:
In all cases, if writing is enabled (write_enable_sel_r == 2'b1), then reset accum_result. Otherwise, keep accumulating.

3. Another section of datapath is responsible for calculating the SRAM read addresses for different phases of matrix operations, particularly during matrix multiplication and accumulation. My code dynamically adjusts the read addresses based on control signals (Q_K_read, S_V_read, and input_read_addr_sel) and counters (counter_for_S, temp_count_var_I_count). The datapath calculates read addresses based on whether it is reading from matrix I, scratchpad memory for S, or scratchpad memory for V. The code dynamically calculates the read addresses for both input SRAM (I) and scratchpad SRAM based on the state of the computation:

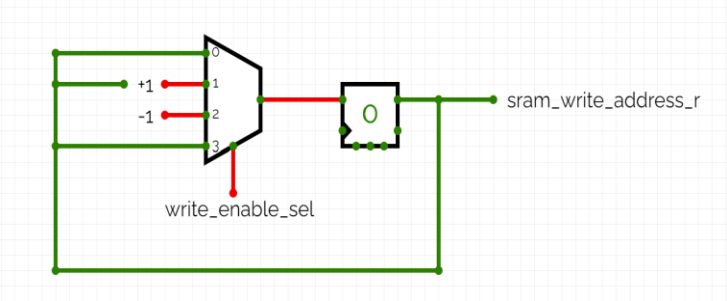
- **Reading from Input SRAM (Matrix I):** If Q_K_read == 0 and S_V_read == 0, the system reads from matrix I. The address increments sequentially unless it reaches the end of a row or column, in which case it resets or adjusts accordingly.
- **Reading from Scratchpad SRAM for Matrix S:** When Q_K_read == 1 and S_V_read == 0, the system reads from matrix S stored in scratchpad memory. The address increments until it reaches the end of a row or column, at which point it resets or adjusts based on conditions.
- **Reading from Scratchpad SRAM for Matrix V:** When both Q_K_read == 1 and S_V_read == 1, it reads from matrix V in scratchpad memory. The address adjusts based on counters (counter_for_S) to ensure correct row/column traversal.

4. This part of datapath is responsible for calculating the read addresses for SRAM W (which stores matrix weights) and SRAM result memory (which stores intermediate results during matrix multiplication and can be used for calculation of other matrices). The address calculation is based on control signals (Q_K_read, S_V_read, weight_read_addr_sel) and counters. It calculates read addresses based on whether it is reading from matrix W, scratchpad memory for intermediate results, or processing matrix V.

Key Phases:

1. **Reading from Matrix W (Weights)**:
    - When both Q_K_read == 0 and S_V_read == 0, the system reads from matrix W.
    - Based on weight_read_addr_sel:
        - **2'b00**: Reset address to zero.
        - **2'b01**: Increment address until the end of the matrix, then reset to the next section (Q, K, or V).
        - **2'b10**: Hold the current address.
        - **2'b11**: Reset to the first address.
2. **Reading from Result Memory for S**:
    - When Q_K_read == 1 and S_V_read == 0, it reads intermediate results for matrix S.
    - Based on weight_read_addr_sel:
        - **2'b00**: Initialize result memory address.
        - **2'b01**: Increment address until the end of a row, then reset.
3. **Reading from Result Memory for V**:
    - When both Q_K_read == 1 and S_V_read == 1, it reads intermediate results for matrix V.
    - Based on weight_read_addr_sel:
        - **2'b00**: Initialize address for V.
        - **2'b01**: Increment address based on counters, resetting when reaching the end of rows/columns.

5. The register sram_write_address_r stores the address of SRAM to write into. Initially it is -1 (i.e 16`hFFFF) and starts increasing when the write_enable is high and this value is assigned to dut__tb__sram_result_write_address and dut__tb__sram_scratchpad_write_address.



The figure 2 displays the FSM with all the states for proposed design for this project with the signal values for each state are as follows set_dut_ready, temp_count_var_I_count_select, temp_count_z_select, input_read_addr_sel, weight_read_addr_sel, write_enable_sel, Q_K_read, S_V_read,Q_done, V_done,S_done, Z_done.

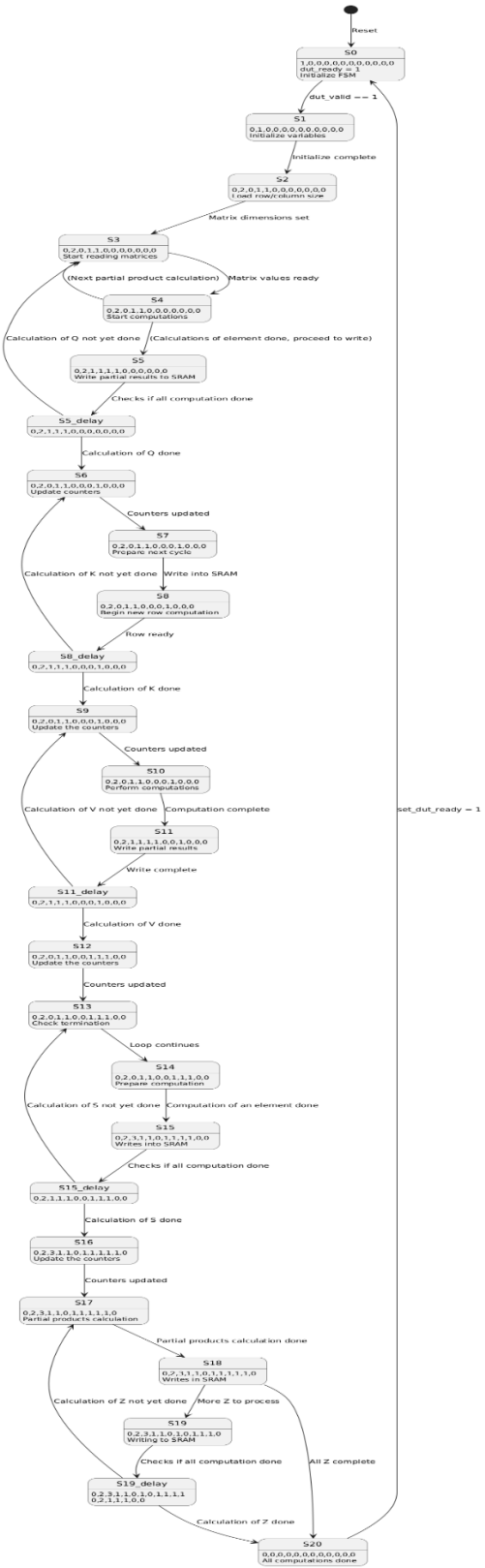


Figure 2: FSM for the implemented design for the project

INTERFACE SPECIFICATION

Detailed Description of Interface

The design implements a hardware accelerator for the Scaled Dot-Product Attention mechanism. The interface consists of several control signals, clock and reset signals, and multiple SRAM interfaces for input, weight, result, and scratchpad memory. Each interface allows reading from and writing to the corresponding SRAMs as part of the matrix computations for Query (Q), Key (K), Value (V), and final attention outputs. The **SRAM interfaces** are used to store and retrieve matrices during the computation of self-attention. The input SRAM holds the input data, the weight SRAM holds the learned weights for Q, K, and V transformations of same dimensions, the result SRAM stores intermediate results, and the scratchpad SRAM is used for temporary storage during calculations. The below table gives information about the interface signals.

Table 1: Information of the interface signals

Signal Name	Width	Function
reset_n	1 bit	Active-low reset signal to initialize the design.
clk	1 bit	Clock signal for synchronous operations.
dut_valid	1 bit	Indicates that valid input data is available for processing.
dut_ready	1 bit	Signals that the DUT is ready to accept new input data.
dut__tb__sram_input_write_enable	1 bit	Enables writing to the input SRAM.
dut__tb__sram_input_write_address	16 bits	Address for writing data to the input SRAM.
dut__tb__sram_input_write_data	32 bits	Data to be written to the input SRAM.
dut__tb__sram_input_read_address	16 bits	Address for reading data from the input SRAM.
tb__dut__sram_input_read_data	32 bits	Data read from the input SRAM.
dut__tb__sram_weight_write_enable	1 bit	Enables writing to the weight SRAM.
dut__tb__sram_weight_write_address	16 bits	Address for writing data to the weight SRAM.
dut__tb__sram_weight_write_data	32 bits	Data to be written to the weight SRAM.
dut__tb__sram_weight_read_address	16 bits	Address for reading data from the weight SRAM.

Signal Name	Width	Function
tb__dut__sram_weight_read_data	32 bits	Data read from the weight SRAM.
dut__tb__sram_result_write_enable	1 bit	Enables writing to the result SRAM.
dut__tb__sram_result_write_address	16 bits	Address for writing data to the result SRAM.
dut__tb__sram_result_write_data	32 bits	Data to be written to the result SRAM.
dut__tb__sram_result_read_address	16 bits	Address for reading data from the result SRAM.
tb__dut__sram_result_read_data	32 bits	Data read from the result SRAM.
dut__tb__sram_scratchpad_write_enable	1 bit	Enables writing to the scratchpad SRAM.
dut__tb__sram_scratchpad_write_address	16 bits	Address for writing data to scratchpad SRAM.
dut__tb__sram_scratchpad_write_data	32 bits	Data to be written to scratchpad SRAM.
dut__tb__sram_scratchpad_read_address	16 bits	Address for reading data from scratchpad SRAM.
tb__dut__sram_scratchpad_read_data	32 bits	Data read from scratchpad SRAM.

Interface Timing Diagram

Below is a description of how signals interact in a typical read/write operation with an example timing diagram:

1. **Reset Phase**: The system is initialized by asserting an active-low reset (reset_n = 0). During this phase, all internal states are cleared.
2. **Idle Phase**: After reset, when no valid input is available (dut_valid = 0), the design remains idle with dut_ready = 1, indicating it is ready to receive new inputs.
3. **Input Read Phase**:
    - Once valid input data is available (dut_valid = 1), the DUT lowers dut_ready = 0, indicating it is processing.
    - The DUT reads from both input and weight SRAM using respective read addresses (dut_tb_sram_input_read_address, dut_tb_sram_weight_read_address). The data retrieved is stored in internal registers.
4. **Computation Phase**:
    - The DUT performs matrix multiplications (e.g., $Q \times KT$) and computes intermediate results such as attention scores.
    - Temporary results may be stored in scratchpad memory during this phase.

5. **Result Write Phase**:
    - After computation, results are written back into result SRAM using write addresses (dut_tb_sram_result_write_address) and write enable signals (dut_tb_sram_result_write_enable = 1).
6. **Completion Phase**:
    - Once all computations are done, and results are stored in result memory, dut_ready = 1, signaling that new inputs can be processed.

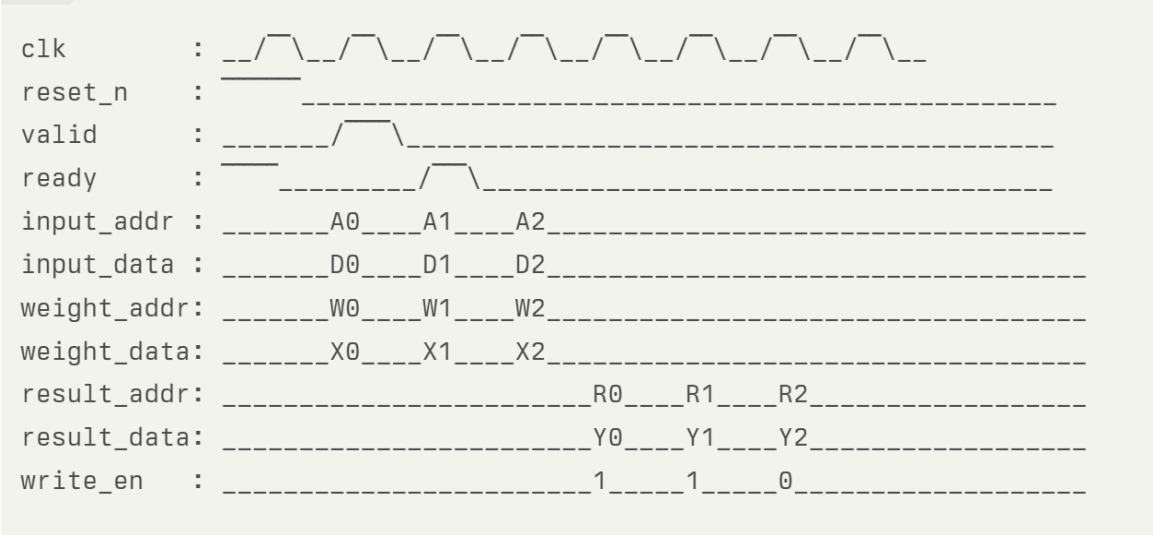


Figure 3: Example timing diagram with sample input address, weighted address, and the data present in those addresses.

In this example:

- The clock (clk) drives all operations.
- The system starts by resetting (reset_n = 0) and then deasserting reset (reset_n = 1) after initialization.
- When valid data is available (valid = 1), addresses are provided for reading from input and weight memories (input_addr, weight_addr). The corresponding data (input_data, weight_data) is retrieved.
- After computation, results are written back into result memory at addresses R0, R1 with corresponding data Y0, Y1 when write enable (write_en) is high.

This timing diagram illustrates how different signals interact during a typical execution cycle of matrix computation and attention score generation in this hardware design.

TECHNICAL IMPLEMENTATION

High-Level Modeling and Results Achieved

The design implements the **Scaled Dot-Product Attention** mechanism, a key component of transformer models, using hardware. The high-level model focuses on the efficient computation of Query (Q), Key (K), and Value (V) matrices through matrix multiplications, followed by attention score calculation and final output generation. The design leverages multiple SRAM interfaces for storing input matrices, weight matrices, intermediate results, and scratchpad memory for temporary storage during computations.

Key results achieved:

- **Efficient Memory Utilization**: The design uses separate SRAM blocks for input, weights, results, and scratchpad memory, ensuring efficient data handling and reuse during matrix operations.
- **Correct Functionality**: The design successfully computes the attention scores and final outputs as expected from the self-attention mechanism. After calculating the matrix values, they are parallelly stored in result SRAM and scratchpad SRAM.

Hierarchy Below High-Level Design

The design is structured into different modules that handle specific tasks within the self-attention computation:

1. **Top-Level Module (MyDesign)**:
    - This module coordinates the overall flow of data between the different components (SRAMs) and controls the state transitions for the various stages of computation.
    - It manages reading inputs from SRAM, performing matrix multiplications, accumulating partial results, writing results back to SRAM, and handling control signals like dut_valid and dut_ready.
2. **State Machine**:
    - A finite state machine (FSM) is used to control the sequence of operations. It moves through states like reading inputs, updating counters, performing matrix multiplications, and writing results to SRAM.
    - The states S0 is an idle state which waits for dut_valid signal to go high, then it changes to next state S1.
    - In state S1 the dimensions of the matrices which are stored in address 0 are read.
    - The S2 updates the counter and the address calculation for input and weight starts at S3.
    - At S4 the partial products are calculated and if the calculation for a Q matrix element is done then it goes to S5 where the data is written into result as well as in the scratchpad SRAMs and moves to S5_delay where it checks if the all the elements of Q are calculated and changes state according to that condition.
    - It reaches S6 to start the calculation of elements of K matrix when all the elements of matrix Q are calculated. Then the same sequence of operations is done from states S6 to S8_delay.

- In the same way the elements of V matrix are calculated and stored in SRAM from states S9 to S11_delay.
- Now to start the calculation of Score matrix (S12 to S15_delay) we require Q and K elements. For this I have chosen to read the Q elements from scratchpad SRAM and K elements from result SRAM.
- In the same way for Z matrix (S17 to S19_delay) the values of S are read from scratchpad and V from result SRAM.
- When all the calculations are done then it moves to S20 where it sets the dut_ready to high and changes state to S0.
- Each state corresponds to a specific stage in the computation process, such as reading Q/K/V matrices or calculating attention scores.

3. **SRAM Interface Modules**:
    - Separate interfaces manage reading from and writing to input SRAM (dut__tb__sram_input_*), weight SRAM (dut__tb__sram_weight_*), result SRAM (dut__tb__sram_result_*), and scratchpad SRAM (dut__tb__sram_scratchpad_*).
    - These modules ensure that data is correctly fetched from memory and written back after computations.

Detailed Implementation

The detailed implementation involves several key components:

1. **SRAM Address Calculations**:
    - Address counters are used to track read/write operations for each SRAM block. For example:
        - sram_read_address_I tracks addresses for reading input data.
        - sram_write_address_r tracks addresses for writing results.
        - Separate counters are used for tracking rows and columns of matrices during matrix multiplications.
2. **Matrix Multiplication Logic**:
    - Matrix multiplication is performed by accumulating partial products in each clock cycle. The FSM controls when to start/stop accumulation based on the current state.
    - For example, in state S4, partial products are accumulated until all elements of a row/column have been processed.
3. **Attention Score Calculation**:
    - Once Q and K matrices are computed, the score matrix $S=Q\times KT$ is calculated by multiplying Q with the *KT*.
    - This score is then scaled by dividing by *dk* before being passed through a softmax function (not shown explicitly in this code but typically part of the attention mechanism).
4. **Control Signals**:
    - The control signals dut_valid and dut_ready are used to manage communication between the DUT and external test fixtures.
    - When dut_valid is asserted, the DUT starts processing data from SRAM; once processing is complete, it asserts dut_ready, indicating that new inputs can be processed.

5. **State Transitions**:
    - The FSM transitions between states based on conditions like whether all rows/columns have been processed or whether a particular matrix multiplication has been completed.
    - For example, in state S5 (write to result SRAM), if all rows/columns have been processed, the FSM moves to S6 (start K calculations).
6. **INT_MUL**:
    - This module used for arithmetic operations such as matrix multiplications to ensure precision in calculations.
    - The accum_result variable computes the cumulative sum of the partial products and mac_result_z gives result of the multiplication of 2 variables.
7. **Data path and Control path separation**:
    - In this design, the **data path** and **control path** are clearly separated to ensure efficient processing and manageability:
    - **Data Path**: The data path handles the actual computations, such as matrix multiplications for Query (Q), Key (K), and Value (V) matrices, accumulation of partial sums, and the final attention score calculations. This involves the flow of data between various SRAM modules (input, weight, result, scratchpad) and registers that hold intermediate results during computation.
    - **Control Path**: The control path is responsible for managing the flow of operations. It includes a finite state machine (FSM) that orchestrates the sequence of actions, such as reading from SRAM, performing matrix multiplications, and writing results back to SRAM. The control signals (dut_valid, dut_ready, write_enable_sel, etc.) ensure that each step in the data path is executed in the correct order.
    - This separation allows for efficient parallel processing in the data path while maintaining precise control over the sequence of operations through the control path

VERIFICATION

Approach Used to Verify Correctness

The verification of the hardware design for the Scaled Dot-Product Attention mechanism was conducted through a comprehensive testbench simulation. The approach involved simulating the design under various test cases to ensure that the implementation correctly performed matrix multiplications and attention score calculations, matching expected outputs.

The verification process included the following steps:

1. **Testbench Setup**:
    - A dedicated testbench was created to simulate the design and provide stimulus to the hardware. The testbench supplied input data (Q, K, V matrices) and weights from memory files (proj_int_test_self_attn_X_input.dat, proj_int_test_self_attn_X_weight.dat), and it compared the results generated by the hardware with expected results stored in result files (proj_int_test_self_attn_X_result.dat).
2. **Multiple Test Cases**:
    - The testbench ran multiple test cases, each testing different matrix dimensions and values. For example, test cases covered various matrix sizes such as [2x4], [2x2], [8x2], and [1x8], ensuring that the design could handle different input configurations.
    - Each test case involved reading input matrices from memory, performing matrix multiplications for Q, K, and V, calculating attention scores, and generating final outputs.
3. **Result Comparison**:
    - After each computation, the hardware-generated results were compared against expected results stored in memory. The comparison was done for each stage of the computation: Query (Q), Key (K), Value (V), Score (S), and Attention (Z) matrices.
    - The testbench printed detailed logs indicating whether each result matched the expected value. For example:
        - **Query Matrix Check**: "Result MATCH: expected_result = 45, dut_result = 45"
        - **Key Matrix Check**: "Result MATCH: expected_result = 99, dut_result = 99"
        - **Value Matrix Check**: "Result MATCH: expected_result = 81, dut_result = 81"
        - **Score Matrix Check**: "Result MATCH: expected_result = 16483, dut_result = 16483"
        - **Attention Matrix Check**: "Result MATCH: expected_result = 2732156, dut_result = 2732156"
4. **Pass/Fail Criteria**:
    - For each test case, all intermediate results were checked for correctness. If all results matched the expected values, the test case was marked as a pass.

- The final pass percentage was calculated based on the number of successful comparisons out of the total number of checks. For instance, one simulation reported a **100% pass rate**, with all 217 result checks passing successfully.

5. **Timing and Performance Metrics**:
    - The testbench also measured performance metrics such as total simulation time and cycle count for each test case.

This verification approach ensured that the hardware implementation was functionally correct across a variety of input sizes and configurations. The detailed result comparisons at each stage of computation provided confidence in the accuracy of the design.

Summary of Verification Results

- **Number of Test Cases**: 4
- **Total Number of Result Checks**: 217
- **Pass Percentage**: 100%

The verification process confirmed that the design correctly implemented the Scaled Dot-Product Attention mechanism, with all test cases passing successfully without errors.

RESULTS ACHIEVED

Throughput

The design was optimized for throughput by leveraging matrix multiplications for Query (Q), Key (K), and Value (V) matrices. This parallelism of accessing elements from 2 SRAMs allowed the design to process multiple elements simultaneously, significantly improving the overall computation speed of the Scaled Dot-Product Attention mechanism. The throughput of the design is directly influenced by the clock frequency and the number of cycles required for each matrix operation. Based on the simulation results, the total cycle count for a complete attention computation was **3,384 cycles** with a total simulation time of **16,920 ns**.

Area

The area of the design is determined by the number of logic cells and memory blocks used in the hardware implementation. The key components contributing to area usage include:

- **Matrix multiplication units**: For Q, K, and V matrices.
- **SRAM blocks**: For storing input data, weights, intermediate results, and final outputs.
- **Control logic**: Including state machines and address generation logic.

Using the clock period of **7.12** ns the area achieved was **13790.7701** $ns^{-1}.um^{-2}$

Timing

The timing analysis from simulation reports indicates that the design meets its timing constraints without any major violations. The total simulation time for a complete attention computation was **16,920 ns**, suggesting that the design operates within acceptable timing margins.

Slack

The smallest positive slack is **0.0022 ns**, which occurs in all three max delay reports. Hold check (min delay): Slack (MET) = **0.0354 ns**
Setup check (max delay) from all three max delay reports: Slack (MET) = **0.0022 ns**

Leakage Power

The leakage power reported is approximately **195.57 million units** after optimizations. Leakage power represents the static power consumption when no switching activity occurs in the circuit. While this value is relatively high due to the use of multiple memory elements (SRAMs), it is expected for a design handling complex matrix operation like those found in self-attention mechanisms.

CONCLUSIONS

This project successfully implemented a hardware accelerator for the Scaled Dot-Product Attention mechanism, a key component of transformer models used in natural language processing (NLP) and artificial intelligence (AI). The design focused on efficient matrix multiplications for Query (Q), Key (K), and Value (V) matrices, followed by attention score (S) calculation and final output generation.

Key Results:

1. **Throughput**: The design achieved a total cycle count of **3,384 cycles** for a complete attention computation, with a total simulation time of **16,920 ns**, demonstrating efficient processing suitable for high-performance applications.
2. **Area**: After optimizations such as constant register removal, the design occupied an area of **13790.7701 $ns^{-1}.um^{-2}$** for clock period **7.12 ns**, which is reasonable for a hardware implementation handling complex matrix operations.
3. **Slack**: The design reported smallest positive slack is **0.0022 ns**, indicating that it met timing requirements comfortably under worst-case conditions, ensuring reliable operation at the specified clock frequency. It can be reduced further to 0.0001 but this might not be synthesizable if there are issues with the libraries.
4. **Power Consumption**: The leakage power was measured at **195.57 million units**, which is expected given the use of multiple memory elements (SRAMs) and mac units. Energy efficiency was improved through parallel processing and optimized memory management.

Summary

The project achieved its goal of implementing an efficient hardware accelerator for the Scaled Dot-Product Attention mechanism. The design demonstrated significant improvements in throughput through parallel matrix multiplications and efficient data storage using SRAM. It met timing constraints and showed reasonable area and power consumption, making it suitable for integration into larger transformer-based AI systems.

FUTURE WORK

Future work on this hardware accelerator for the Scaled Dot-Product Attention mechanism can focus on several key areas to further optimize performance and scalability:

1. **Power Optimization**:
    - **Dynamic Power Reduction**: While the current design focuses on efficient memory usage and parallel processing, further optimizations can be made to reduce dynamic power consumption. Techniques such as clock gating, and reducing unnecessary switching activity could be implemented to minimize power usage during active computation. Additionally, employing lower supply voltages or multi-threshold voltage (multi-Vt) cells could help reduce power consumption without sacrificing performance.
    - **Energy-Efficient Arithmetic**: Optimizing the arithmetic units (e.g., mac units) used for matrix multiplications can also contribute to lower energy consumption. By reducing the number of operations or using approximate computing techniques, the overall energy footprint of the design could be reduced.
2. **Scalability for Larger Transformer Models**:
    - As transformer models continue to grow, supporting larger input sequences and higher-dimensional matrices will be critical. The current design could be scaled by increasing the size of SRAM modules and optimizing the memory hierarchy to handle larger matrices efficiently.
    - Additionally, pipelining and parallelism can be extended to support multi-head attention with more attention heads and layers, ensuring that the hardware can process large-scale transformer models without significant performance degradation.
3. **Optimized Calculation of QKT / √ dk**:
    - **Square Root Calculation**: To optimize the calculation of QKT / √ dk, DesignWare IP for square root computation can be integrated into the hardware. DesignWare provides efficient hardware implementations for complex mathematical functions like square roots, which can significantly improve both performance and accuracy compared to custom implementations.
    - **Division by Continuous Subtraction**: Since division is not synthesizable in traditional hardware design flows, a possible approach is to implement division using an iterative method like continuous subtraction or restoring/non-restoring division algorithms. These methods can compute the division by subtracting the divisor from the dividend iteratively until convergence. This approach can be pipelined to ensure that it does not become a bottleneck in the overall computation pipeline.

By focusing on these areas, future iterations of this hardware design will be able to handle larger transformer models more efficiently while reducing both dynamic power consumption and overall energy usage.

SYMBOL EXPLANATION

Symbol	Meaning
Q	Query matrix, derived from the input sequence via a linear transformation.
K	Key matrix, derived from the input sequence via a linear transformation.
KT	Transpose of key matrix.
V	Value matrix, derived from the input sequence via a linear transformation.
S	Attention score matrix, computed from Q and K.
Z	Final output matrix after applying attention scores to V.
dk	Dimensionality of the Key vectors.
Q**KT*	Dot-product of Query and transposed Key matrices.
softmax	Softmax function applied to scale attention scores into probabilities.
WQ, WK, WV	Weight matrices used to transform input sequences into Q, K, and V matrices.
SRAM	Static Random-Access Memory used for storing intermediate results and matrices.
FSM	Finite State Machine managing control flow in the hardware design.