Final Project Report

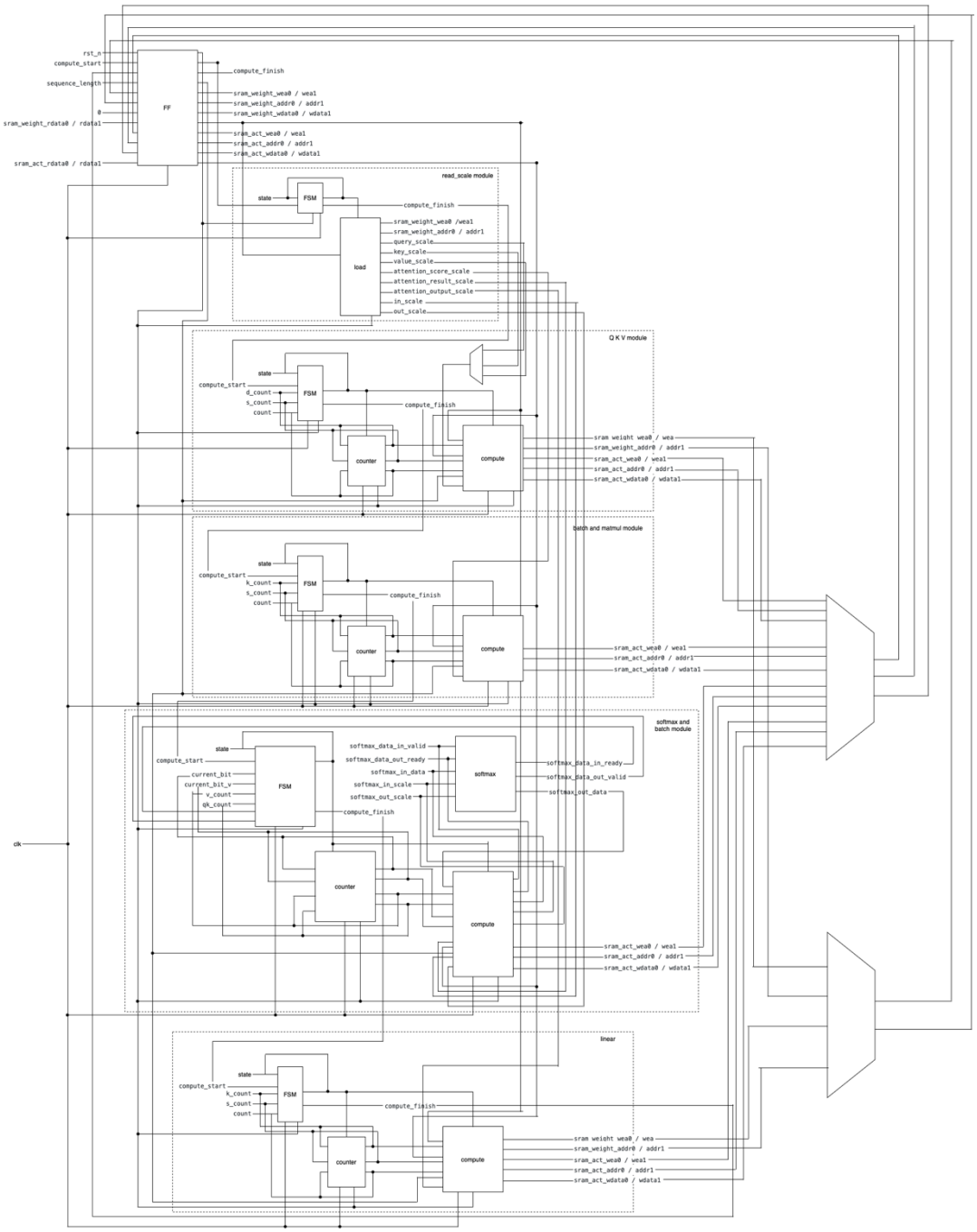
Student ID: 110062322 / 110062335 Name: 謝棻伃 / 許芳語

1. Design concept:

- block diagrams

Input -> FC1 output

The diagram below represents the block diagram from input to fc1.



At the beginning, the SRAM data first passes through a flip-flop before being

forwarded to the next modules. The SRAM input receives the output from the module, which has also passed through a flip-flop.

Read Scale Module

Next, we enter the read scale stage. In this module, all the required scales used in the input to fc1 phase are read. Once the computation is completed, the compute finish signal is sent to the qkv module's compute start. The read scales are then passed to the corresponding modules, enabling them to perform requantization.

QKV Module

In the qkv module, a counter is used to determine the memory addresses to be read and to track when state transitions should occur. Data is read from the specified addresses, computations are performed, and the results are temporarily stored in activation SRAM. After processing all QKV computations, the compute finish signal is set to 1 and passed to the next module’s compute start.

Batch MatMul and Normalize

After completing the linear transformation and transpose in the QKV module, the process moves to the batch matrix multiplication (MatMul) and normalization phase. Here, the previously stored data in activation SRAM is used for computation. The computed results are then stored back into SRAM, serving as temporary storage for subsequent operations. Similar to earlier stages, the compute finish signal is connected to the next module’s compute start.

Softmax and Batch MatMul

In this stage, Softmax is introduced in addition to batch MatMul, making this module slightly different from the previous ones. A counter is still used to track the current memory address to be read. The SRAM output is read, and based on the current state, the data is either forwarded to Softmax for calculation or used in batch MatMul.

Once computations are complete, the results are stored back into SRAM, as the next module will require these values. The compute finish signal is also passed to the following module.

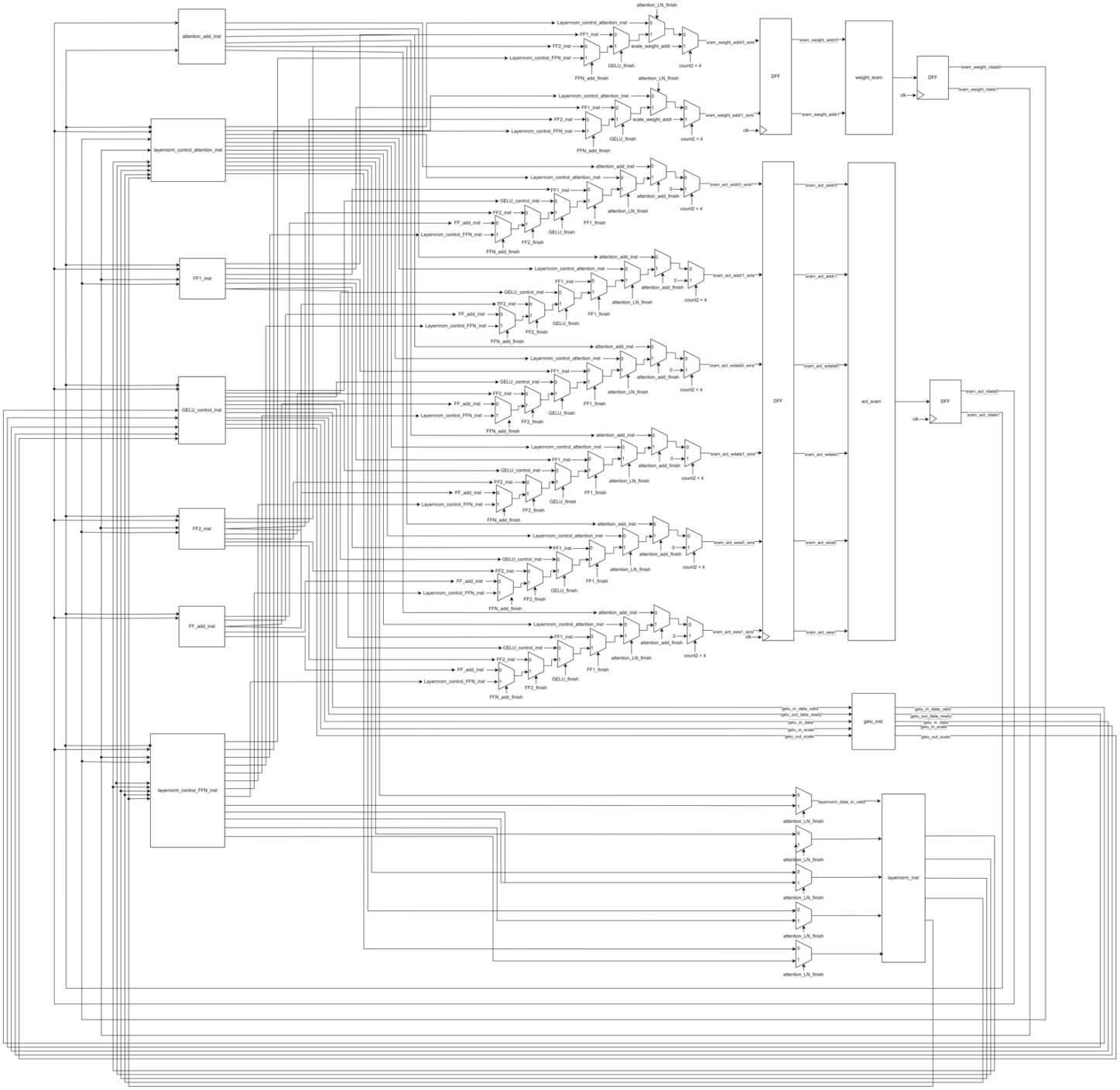
FC1 output -> output

Our approach follows a step-by-step computation, meaning each step must be fully completed before proceeding to the next. The overall signal propagation is determined by the finish signal output from each instance,

which indicates the current progress. Based on this information, the corresponding signals are passed into SRAM, LayerNorm, and GELU. Additionally, before data is written to SRAM, it first passes through a flip-flop (FF), and similarly, the output data goes through an FF before being forwarded to individual instances.

Step 1: Add & LayerNorm after Attention

The first stage involves Add & LayerNorm following Attention. Since Add



does not require weights, the only signals passed out are those related to reading from activation SRAM (act SRAM).

Step 2: LayerNorm

In the second stage, LayerNorm is applied. As discussed in the forum, the signal should not pass through an FF, so it is directly connected to the LayerNorm control instance. The LayerNorm instance (layernorm_inst) utilizes the control signals and input data to perform calculations. The output is then stored in SRAM. However, since any signal entering SRAM must go through an FF, the LayerNorm output must also pass through an FF before being stored.

Step 3: Feedforward Layer

The Feedforward Layer consists of two linear layers and one GELU activation.

1. The linear layers are adapted from previous assignments, where weights and activations are read based on their respective addresses (addr).
2. The values are then multiplied, accumulated, and finally quantized.

Step 4: GELU Activation

The GELU control instance mainly manages the valid, ready, and act SRAM address signals, ensuring that GELU instance (gelu_inst) receives the correct inputs.

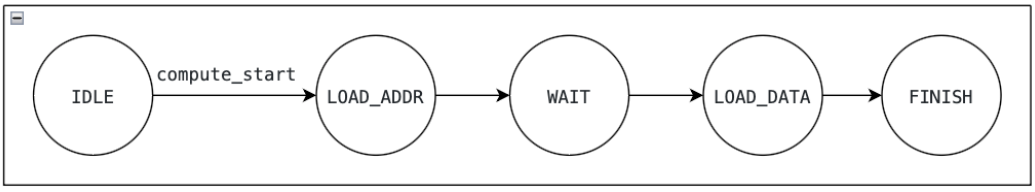
Step 5: Add & Norm after Feedforward Layer

The final step is Add & LayerNorm after the Feedforward Layer, which follows the same process as Attention's Add & LayerNorm, except for a change in the address used for reading data.

- FSM

Input -> FC1 output

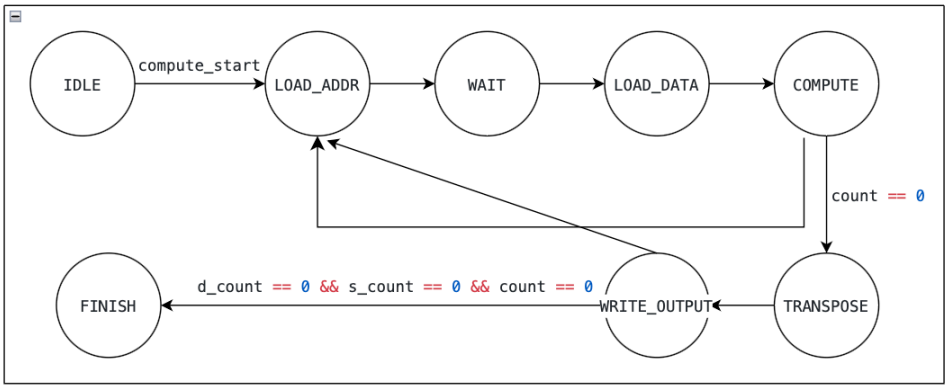
Read scale



The diagram above represents the Finite State Machine (FSM) of the Read Scale Module. Initially, the system remains in the IDLE state, waiting for compute start to be set high. Once compute start becomes 1, the FSM transitions to the LOAD_ADDR state, where it loads the address of the stored scale into SRAM. It

then moves to the WAIT state, pausing for one clock cycle to allow SRAM to return the data. After receiving the data, the FSM transitions to the LOAD_DATA state, where the scale values are loaded. Finally, the FSM enters the FINISH state, completing the process.

QKV module

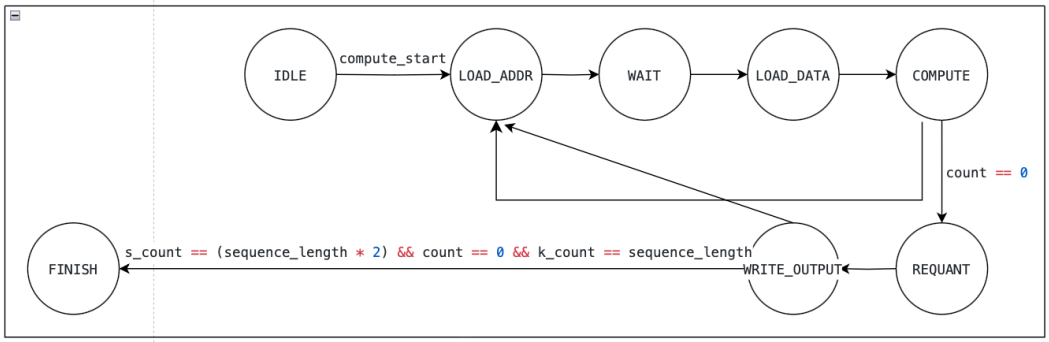


The diagram above represents the Finite State Machine (FSM) of the QKV Module. Initially, the system stays in the IDLE state, waiting for compute start, which is set to 1 only after the Read Scale Module completes. Once compute start becomes 1, the FSM transitions to the LOAD_ADDR state, where it determines the addresses of the weights and activations to be read based on the counter's value.

Next, the FSM enters the WAIT state, pausing for one clock cycle before moving to the LOAD_DATA state, where the previously specified weights and activations are loaded into local storage. In the following cycle, it transitions to the COMPUTE state, where computations are performed using the loaded data. When the count variable reaches 0, it indicates that one output value has been fully computed. At this point, the FSM moves to the TRANSPOSE state, where additional operations such as adding bias, scaling, clamping, and transposition are performed.

After transposition, the FSM enters the WRITE_OUTPUT state, where the computed output is stored in SRAM. If all variables are 0, it means all computations have been completed, and the FSM transitions to the FINISH state, marking the end of the process.

Batch matmul and normalize

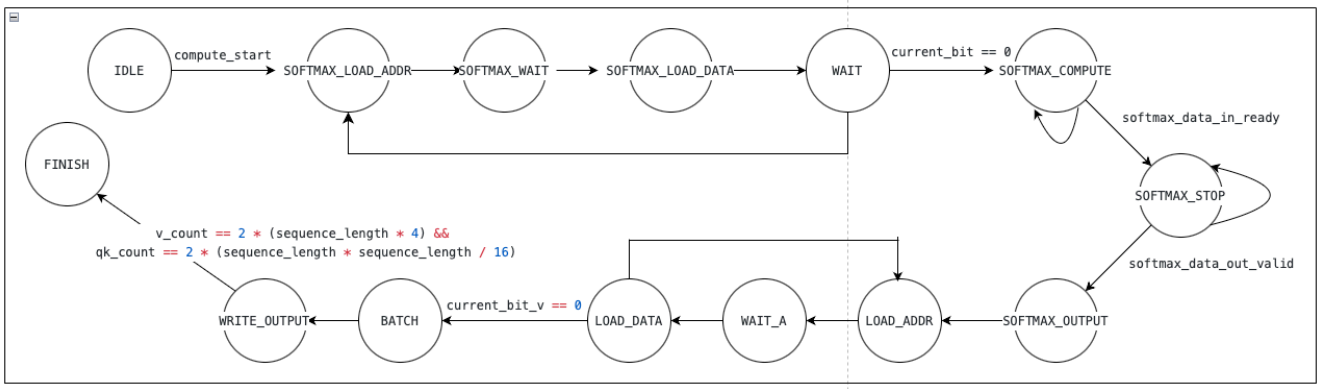


The diagram above represents the Finite State Machine (FSM) for this part of the process, which is quite similar to the previous one. There are only two key differences.

The first difference is that after the COMPUTE state, if count = 0, meaning the output is ready, the FSM transitions to the REQUANT state instead of the TRANSPOSE state from the previous part. In this state, the transpose and add bias steps are omitted compared to the previous FSM.

The second difference lies in the condition for entering the FINISH state. Since this process runs using a two-batch approach, the s_count must reach sequence_length * 2 before the FSM can transition to FINISH. Apart from these differences, the overall structure remains the same.

Softmax add batch matmul



The diagram above represents the Finite State Machine (FSM) for Softmax and Batch MatMul, which is more complex compared to the previous FSMs.

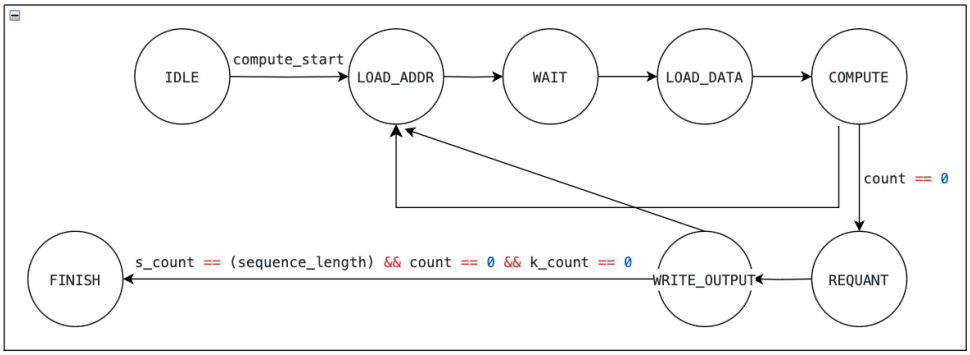
The process begins with the SOFTMAX_LOAD_ADDR state, where the system determines the address from which the input values for Softmax should be read. Once the address is set, the system retrieves data from SRAM, which, as usual, takes one clock cycle to return the data. The FSM then transitions to the SOFTMAX_LOAD_DATA state, where the retrieved values are stored for Softmax computation.

Next, the WAIT state decides whether to proceed to Softmax computation. Since Softmax requires an entire sequence length of data for input, it is possible that the system may not retrieve all necessary data in a single cycle. If current_bit == 0, it indicates that a full sequence length of data has been completely read, allowing the FSM to transition to the SOFTMAX_COMPUTE state to perform the Softmax computation. Otherwise, the FSM returns to SOFTMAX_LOAD_ADDR state to load the next batch of data.

Upon entering the SOFTMAX_COMPUTATION state, the system sets softmax_data_in_valid = 1, signaling that the input data is ready for processing. Once softmax_data_in_ready is received, the FSM moves to the SOFTMAX_STOP state, where the prepared data is passed to Softmax, and softmax_data_out_ready is set to 1, indicating readiness to receive the computed results. When softmax_data_out_valid is returned, meaning Softmax computation is complete, the FSM transitions to the SOFTMAX_OUTPUT state, where the computed Softmax output is stored locally for subsequent computations.

After completing the Softmax states, the FSM proceeds similarly to previous stages, where it reads weights and biases from SRAM and performs the necessary computations.

Linear

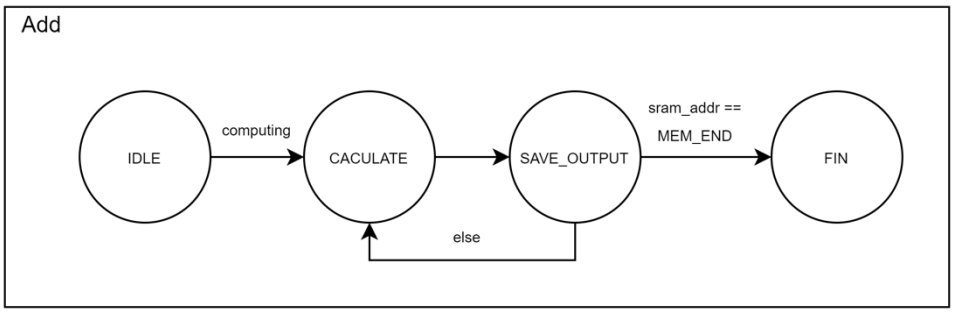


Similar to the Batch MatMul and Normalize FSM, the diagram above represents the Linear FSM. The only change is the condition for transitioning to the FINISH state. Since there is no batching in this process, s_count only needs to reach sequence_length instead of sequence_length * 2.

Additionally, in the REQUANT state, the computation includes an extra step to add bias, while all other operations remain unchanged.

FC1 output -> output

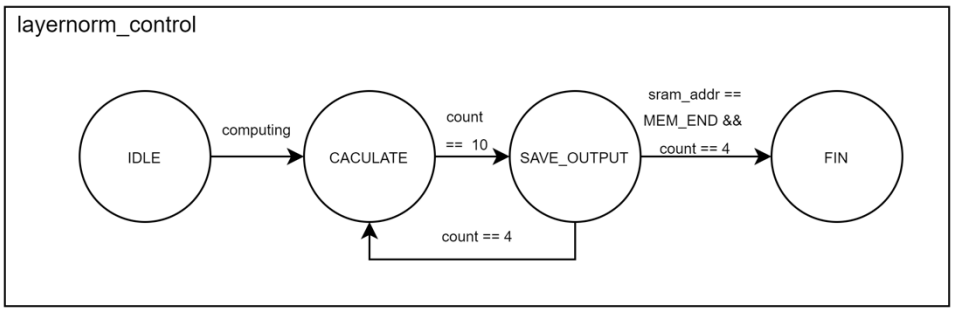
Add



Initially, upon receiving the computation start signal, the FSM sets computing to active and transitions to the CALCULATE state. During CALCULATE, it takes one cycle to read the two activation values that need to be added. Since this step requires only one cycle, the FSM immediately moves to SAVE_OUTPUT, where the computed sum is stored in SRAM.

Because only one address of data needs to be stored, the FSM moves to the next state after just one cycle. If all computations are completed, it transitions to the FIN state; otherwise, it proceeds back to CALCULATE for the next operation.

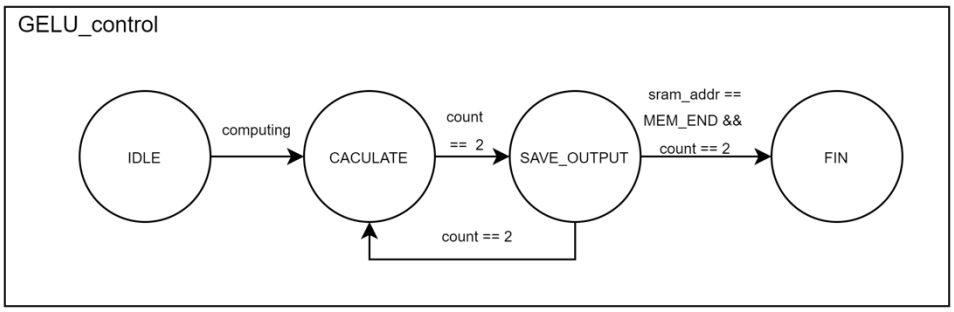
Layernorm control



The FSM of LayerNorm follows the same states as the Add Module but with some differences. Since LayerNorm's input needs to be fed in four separate cycles, and considering SRAM's latency, a counter is used to track the process.

When the counter reaches 10, it indicates that LayerNorm input processing is complete, and the FSM transitions to the SAVE_OUTPUT state, where the computed LayerNorm output is stored in SRAM. The output is generated over four cycles, so once the counter reaches four, the FSM returns to the CALCULATE state for the next computation. If all calculations are complete, the FSM transitions to the FIN state.

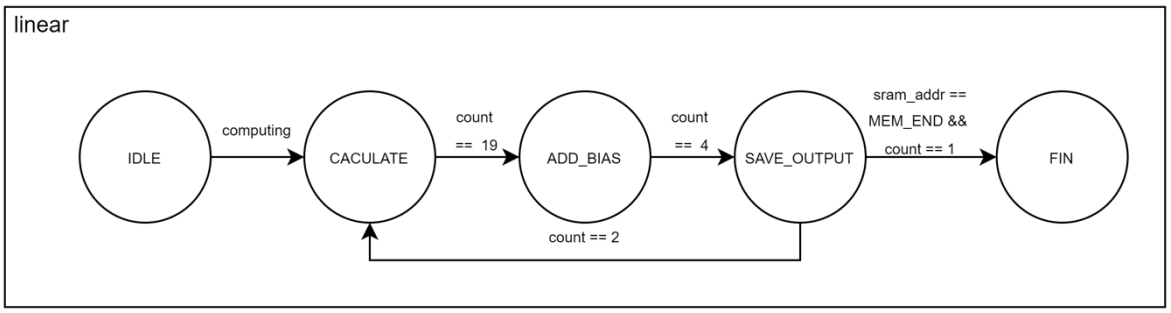
GELU control



The FSM of GELU follows the same states as LayerNorm and Add, with slight modifications. Since GELU processes only one address per input cycle, and considering SRAM's latency, a counter is used to track progress.

When the counter reaches the designated value, it indicates that GELU input processing is complete, and the FSM transitions to the SAVE_OUTPUT state, where the computed GELU output is stored in SRAM. The output follows the same cycle count as the input, so when the counter reaches two, the FSM returns to the CALCULATE state for the next computation. If all calculations are completed, the FSM transitions to the FIN state.

Linear



For FF1, computing a single element requires reading 8 addresses, which takes 4 cycles. Due to SRAM latency, the FSM transitions to the next state when the counter reaches the designated value.

For FF2, computing a single element requires reading 32 addresses, which takes 16 cycles. As a result, the FSM transitions to the next state when the counter reaches 19.

The next state is ADD_BIAS, where the bias is retrieved and added to the accumulated sum. Following this, the FSM transitions to SAVE_OUTPUT, which operates in the same manner as the previous control logic.

2. Result

Item	Description	Unit

RTL simulation	PASS	---
Gate-level simulation	PASS	---
Gate-level simulation clock period	27	ns
Gate-level simulation latency	687452	cycles
Total cell area	1916236.901247	um^2

3. Contribution

Item	Student1	Student2
Architectural design	50%	50%
Coding	50%	50%
Report writing	50%	50%

4. END_CYCLE

END_CYCLE	1000000