Matrix Multiplication for Scaled Dot-Product Attention Equation in Transformer

Harry Hsiao

Abstract
The project implements a critical component of the Transformer architecture: the Scaled Dot-Product Attention mechanism. The design focuses on the hardware realization of the mathematical expression:

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

The implementation performs matrix multiplications between input embeddings and weight matrices to generate the Query (Q), Key (K), and Value (V) matrices, followed by calculating the score matrix and the final attention output.

The design interfaces with three SRAM modules for storing inputs, weight parameters, and results, using a handshake protocol for control flow.

The entire code implements the following five steps:

1. $Q = I * W^Q$

2. $K = I * W^K$

3. $V = I* W^V$

4. $S = Q * K^T$ (Transpose)

5. $Z = S * V$

1. Introduction

This design implements a deeply pipelined architecture to simultaneously fetch data from SRAMs and perform five-matrix multiplications for the attention mechanism in a transformer. Key features and components of this design include:

- A sub-module dedicated to performing accumulated matrix multiplications.
- A datapath designed to calculate subsequent read addresses efficiently.
- Six registers to ensure the correct execution of the multiplication function.
- A finite-state machine was developed to manage data flow and control operations.
- Four counters were implemented to regulate timing, including write operations and state transitions.

- Hardware used in the design
  1. A 9-state Moore Machine that manages datapath
  2. Four counters that manage the states of finite state machine
  3. Four registers that record the dimension of the matrix
  4. Four registers that calculate the read and write address
  5. Six registers that document the necessary address of the calculated matrix
  6. A sub-module named integer_accumulated_matrix_multiplication to perform matrix multiplication

- Key innovations
  1. Deep pipelined design that only stalls the calculation in a one-cycle writing state

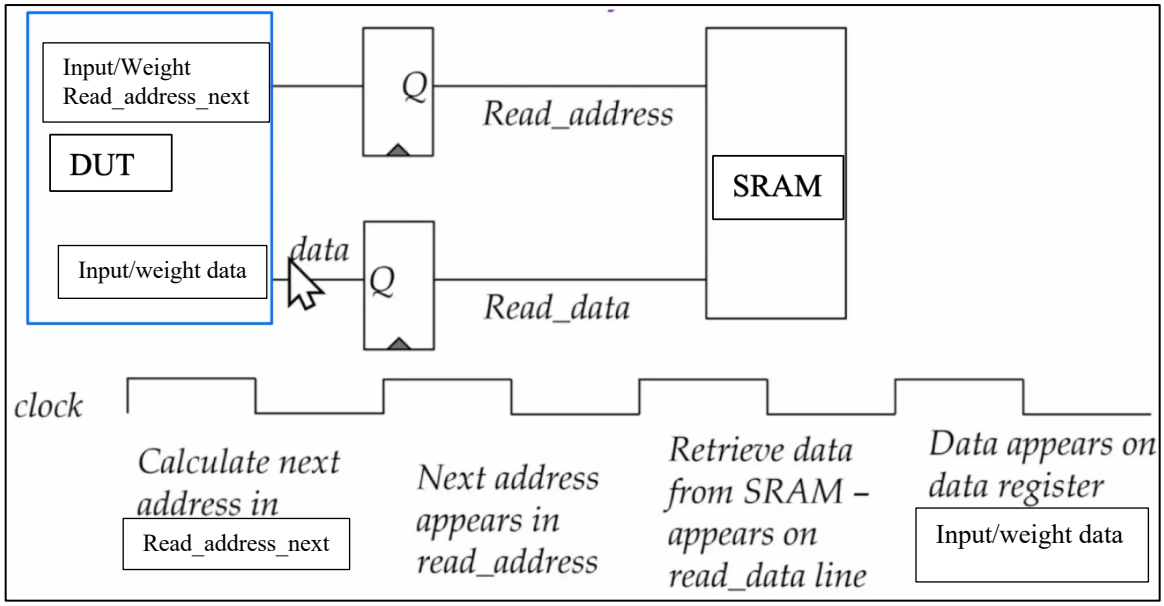
- Results & Achievement
  1. Pass all for 4 test cases with 217 result data
  2. Total logic (combinational + sequential) area = 9766.72um^2
  3. Total cycles for 4 test cases with 217 result data = 1213
  4. Total delay for 4 test cases with 217 result data = 6671.5ns
  5. Performance 1/(delay x area): 15.2088 ns^{-1}.um^{-2}

- Table content
  - Introduction (this page)
  - Micro-Architecture
  - Interface Specification
  - Technical Implementation
  - Verification
  - Results Achieved
  - Conclusion
  - Appendix: Address allocation

2. Micro-Architecture

2.1 High-level architecture

DUT and SRAM

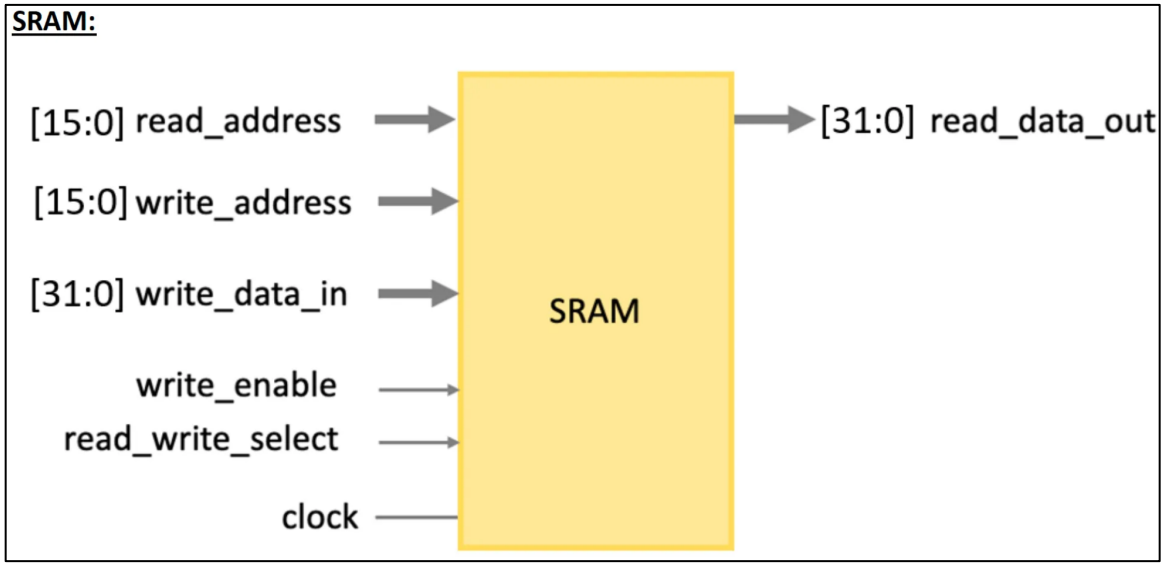


Datapath of address and data:

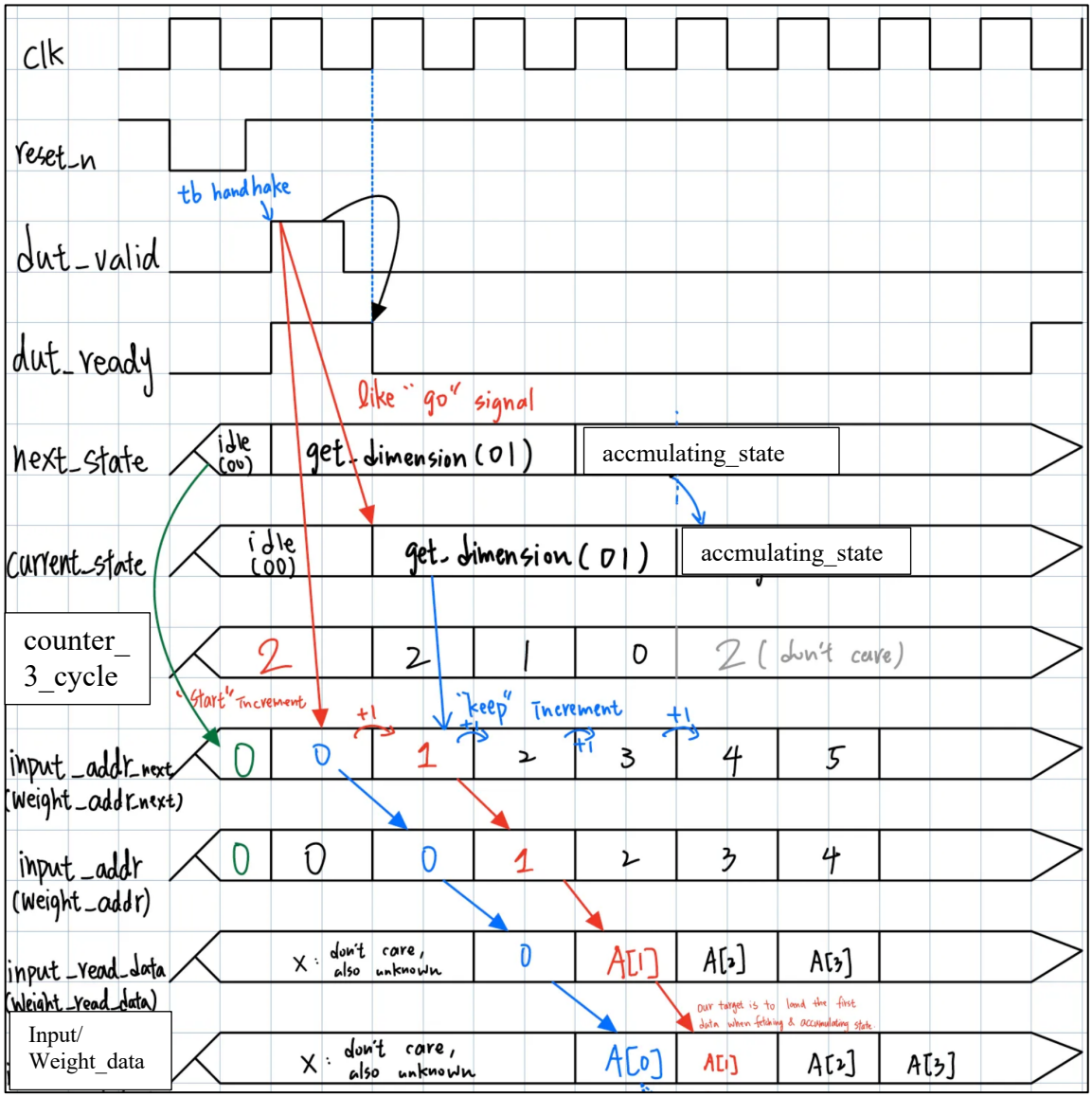
Input/Weight_read_address_next -> Read_address -> Read_data -> Input/weight data

To ensure the correct timing for performing multiplication, I set up a three-cycle counter for the S1, S5, S6, S7, and S8 states in the FSMD (see below). These states last three cycles each. Combined with one cycle at S0 and S4, the desired value aligns perfectly with S2 (the accumulation state). This ensures the first correct data is available precisely when needed for the calculation.

SRAM:



SRAM Interface

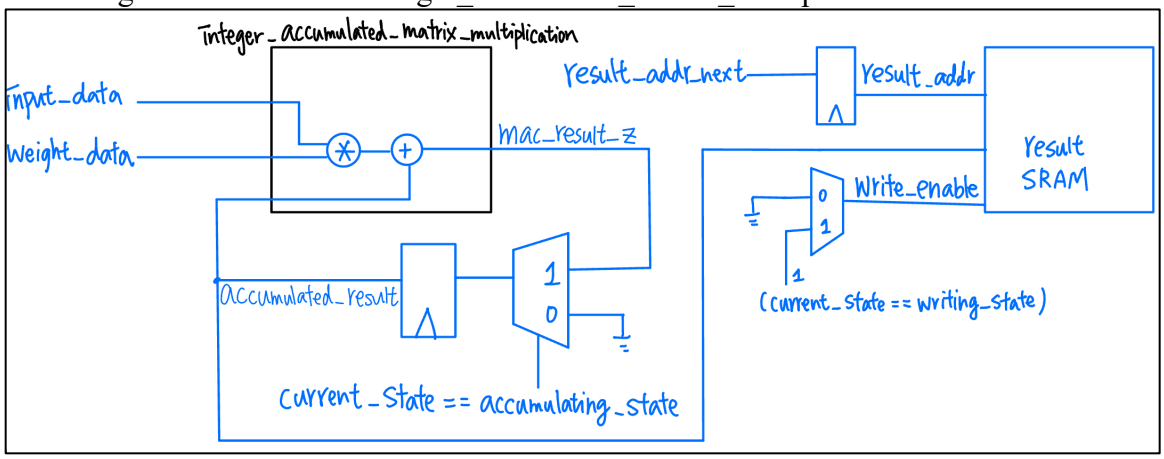


This is the desired timing diagram.

One of the targets in this design is to let the first data we want to calculate pop up when the current_state == accumulating_state. Therefore, as mentioned on the above page, get_dimension will last for three cycles, waiting for the first desired data to pop up when switching to accumulating_state(S2). The same concept happens in the setup_multiplication_n_state, n = 5~8(S5~S8) states, where we set up the counter for the next steps/multiplication.

*A[1] is the first desired data in this timing diagram.

Self-designed sub-module: Interger_accmulated_matrix_multiplication

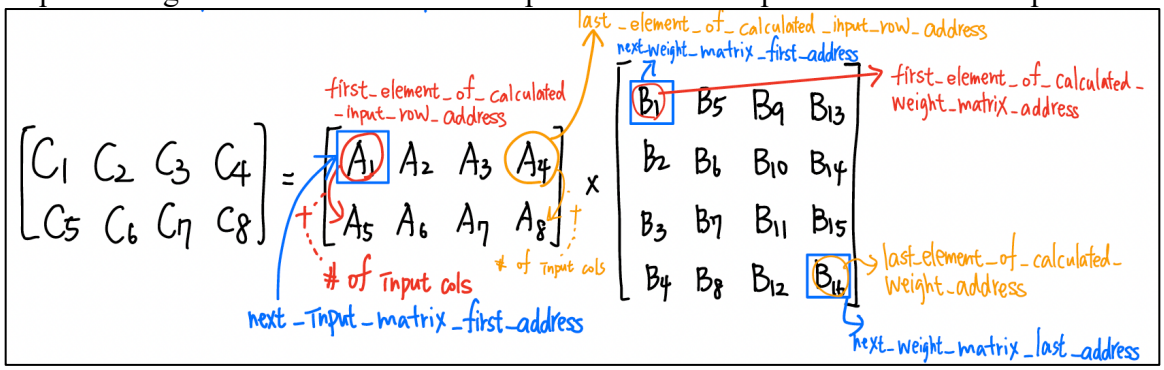


Sub-module that performs matrix multiplication

The circuit continues accumulating results in the accumulating_state. When transitioning to another state, such as the writing_state, the accumulated_result must be cleared to 0 to prepare for the next calculation in the result matrix.

*In the design, I write the result not only in the result SRAM but also in the scratchpad SRAM. This helps us to have higher performance in the last two steps, $S = Q * K^T$ and $Z = S*V$, by fetching input and weight data from the result and scratchpad SRAM simultaneously.

Important registers that document the important address to perform matrix multiplication



C = A x B

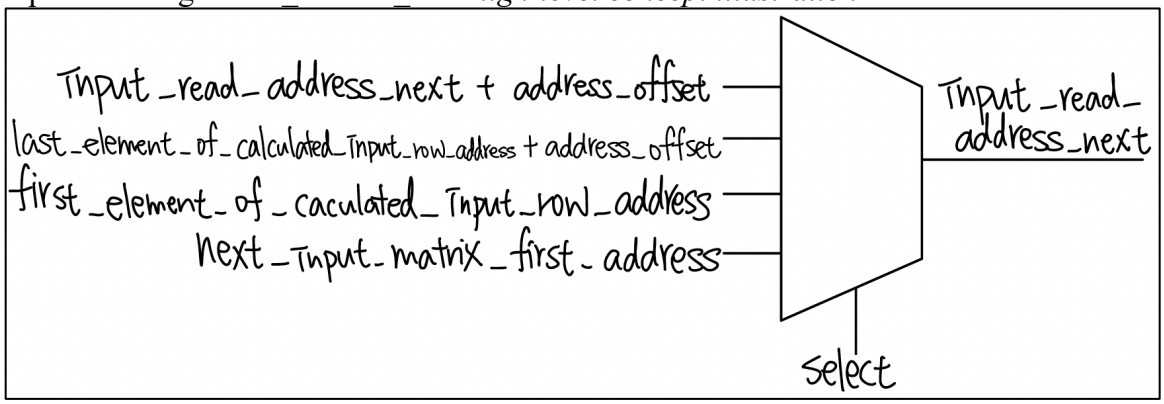
In the number in the right corner, document the address of the element in the matrix. For example, A_7's address is 7 in the input matrix.

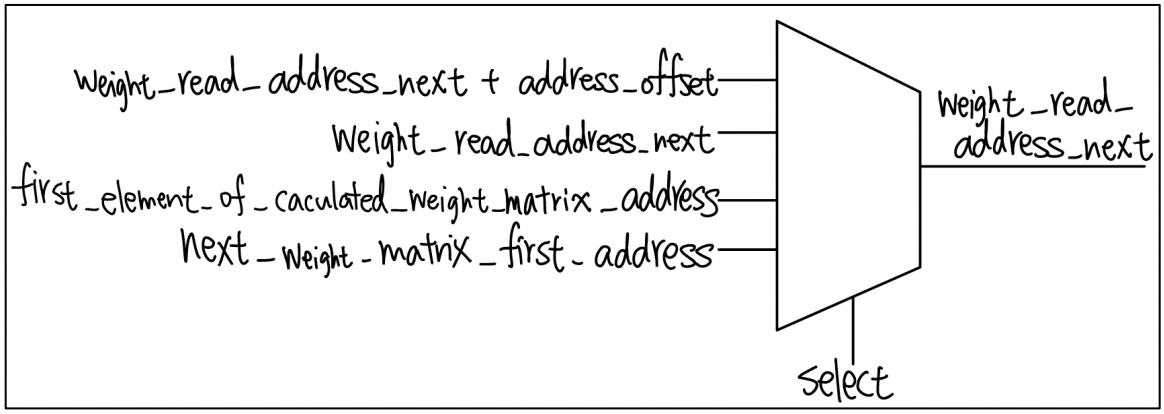
In the designed algorithm, we document some important addresses that help us to correctly update either the input or weight read_address_next that is sent to the SRAM to fetch data. Here is the function of each variable:

- next_input_matrix_first_address: Document the first address of the next input matrix and assign this value to input_read_address_next in the one-cycle next_multiplication_setup_state.

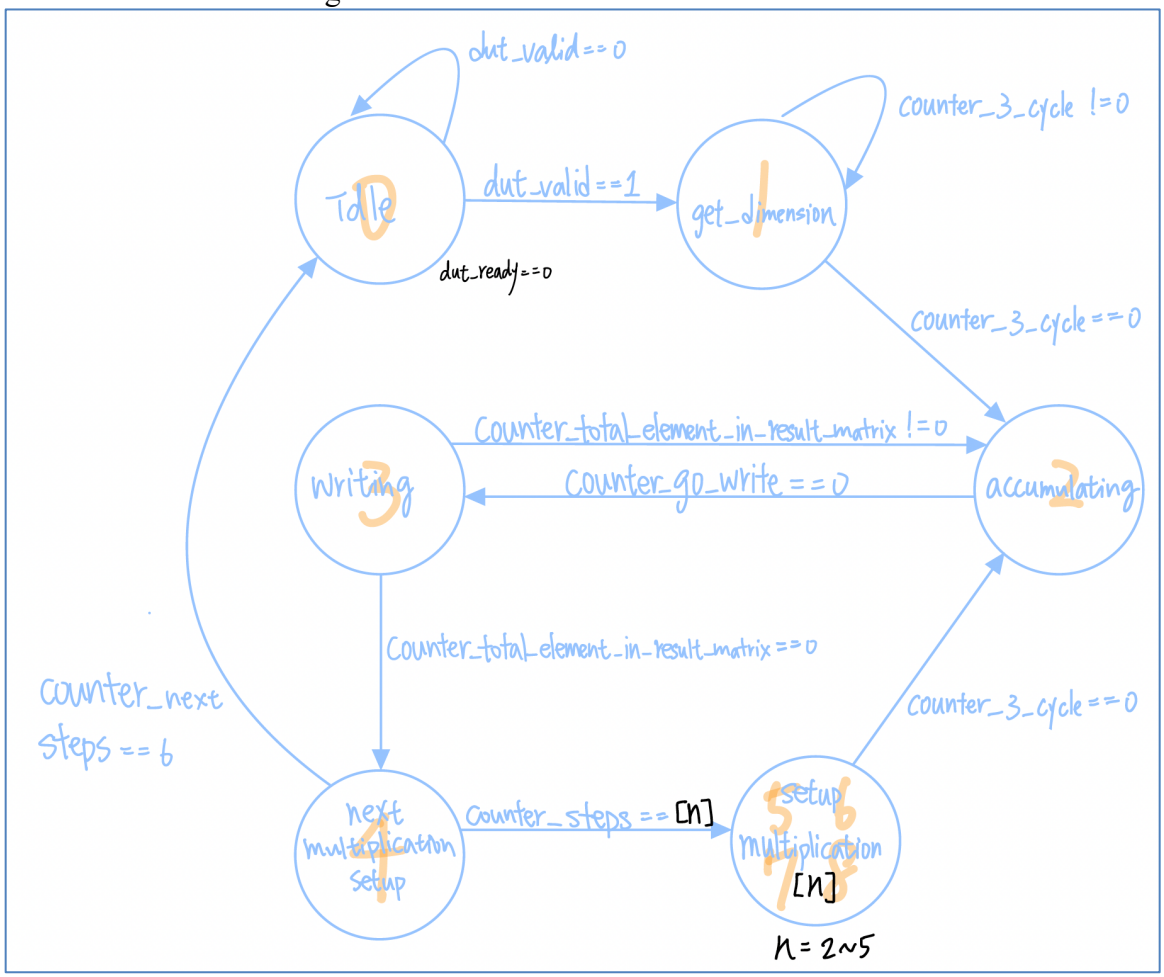
- next_weight_matrix_first_address: Similar function like next_input_matrix_first_address, instead, assign value to the weight_read_address_next.
- first_element_of_calculated_input_row_address: This value helps the input matrix circle back to the first element of the calculated input row, for example, after writing the result of C_1, the input matrix needs to circle back to A_1 again, fetching the A_1~A_4 to perform the calculation with B_5~B_8.
- last_element_of_calculated_input_row_address: This address mainly helps the circuit to detect the timing of the circling back to the first address of the calculated row or implementing the multiplication of the next row. (The algorithm will be explained in the later section.)
- first_element_of_calculated_weight_matrix_address: This value helps the weight matrix circle back to the first address of the calculated weight address. The time needed to circle back is the time we write down the first row in the result matrix (= the time weight_read_address_next is equal to the last address of the weight matrix)
- last_element_of_calculated_weight_matrix_address: This address, combined with the weight_read_address_next, helps the circuit detect the condition that we are going to write the result element in the next row in the result matrix. When this situation happens, we will specially update both input&weight_read_address, not just merely plus a normal address offset (The algorithm in the following section)

input and weight read_address_next *high-level concept illustration*





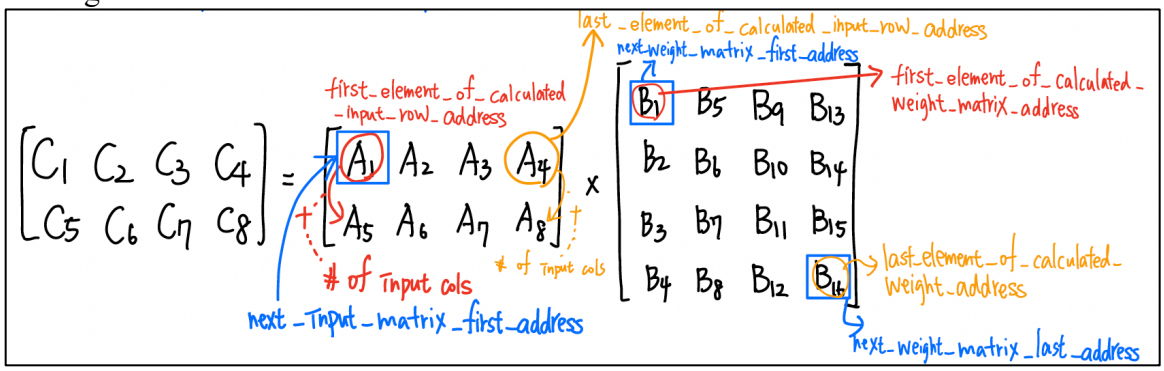
Finite State Machine Diagram:



This is a 9-state Moore Machine, and the function of each state is described as follows:

1. idle: Waiting for dut_valid, which serves as a handshake between the circuit and testbench, our circuit will be initiated one dut_valid signal activates high
2. get_dimension: A state that helps the circuit to land the dimension of the input and weight matrix, including the number of rows and columns. Notice: We only record the dimension information from the first matrix pair, as the dimension of the matrix will be inherited from them afterward due to the natural property of matrix multiplication.
3. accumulating: Keep fetching data from the SRAM and perform matrix multiplication.
4. writing: Write the accumulated result to the result SRAM.
5. next_multiplication_setup: State that sets up the multiple key addresses for the next multiplication, such as the next read address and the first address of the matrix to perform the next multiplication
6. setup_multiplication_[n] (n= 5~8): These states set up the counters for next matrix multiplication to control FSM.

2.2 Algorithm



Hardware algorithmic approach (pseudo code):

```
//input_read_address_next
if (idle state)
        input_read_address_next <= 0;
else if (finish the calculation of the first row in the result matrix && accumulating state)
        start fetching the value in the next row of the input matrix//i.e., A5 ~ A8
else if (finish one result element calculation && accumulating state)
        fetch the value from the same row again
else if (accumulating state)
        increase input_read_address_next with a proper address offset//not necessary = 1
else if( setup next multiplication state)
        assign the first address of the next input matrix to input_read_address_next

//weight_read_address_next
if (idle state)
        weight_read_address_next <= 0;
else if (finish the calculation of the first row in the result matrix && accumulating state)
        circle back to the first address of the weight matrix
else if (accumulating state)
        increase weight_read_address_next with a proper address offset//not necessary= 1
else if (setup next multiplication state)
        assign the first address of the next weight matrix to weight_read_address_next
```

/*The condition above happens when the following condition meets*/

- finish the calculation of the first row in the result matrix: This happens when the weight_read_address_next = last element of the calculated weight matrix address
- finish one result element calculation: This condition happens when the input_read_address_next = last_element_of_calculated_input_row_address

These two conditions might happen at the same time. The first condition takes priority over the second one. Therefore, we need to use the “if else” statement to declare the priority. Additionally, the address offset is not necessarily equal to one due to the transposition of the matrix. This design doesn’t spend extra time to perform the transposition of the matrix K. Instead, it fetches the data in a transposed order by calculating the weight_read_address_next in a transposed way. So, the address offset might not always be one in this design (The following figure illustrates the concept)

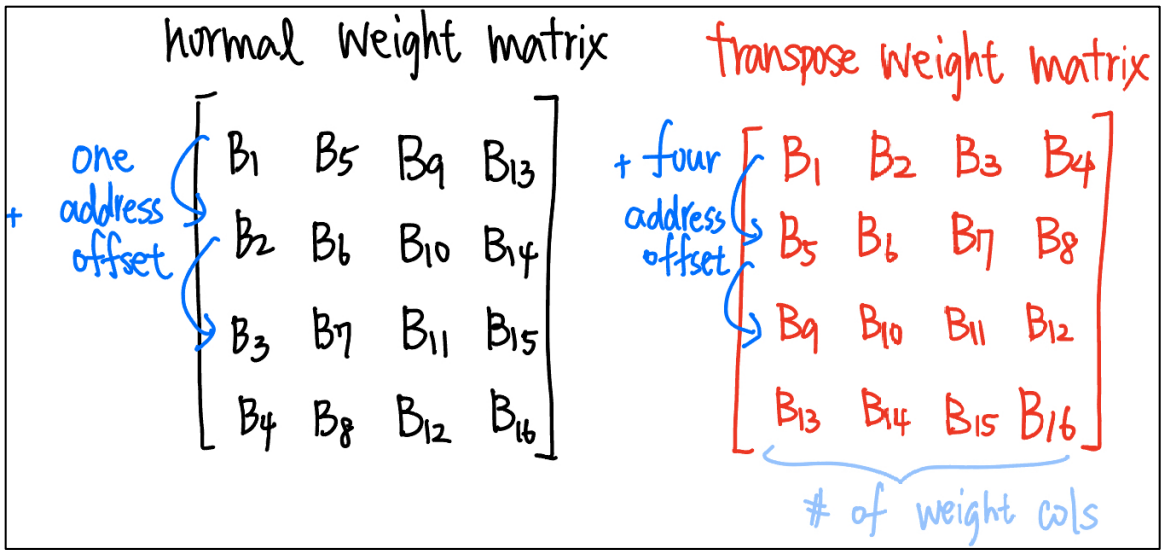


Illustration of different addresses offset due to matrix transpose

Based on the above figure, the address offset will be different when we want to perform matrix multiplication with a transposed matrix. This ACTUALLY happens in step 5(not step 4), where Z = S * V due to the address allocation of the SPEC.

Instead of input and weight read address next, the circuit also updates the value of

- first_element_of_calculated_input_row_address
- last_element_of_calculated_input_row_address
- first_element_of_calculated_weight_matrix_address
- last_element_of_calculated_weight_matrix_address

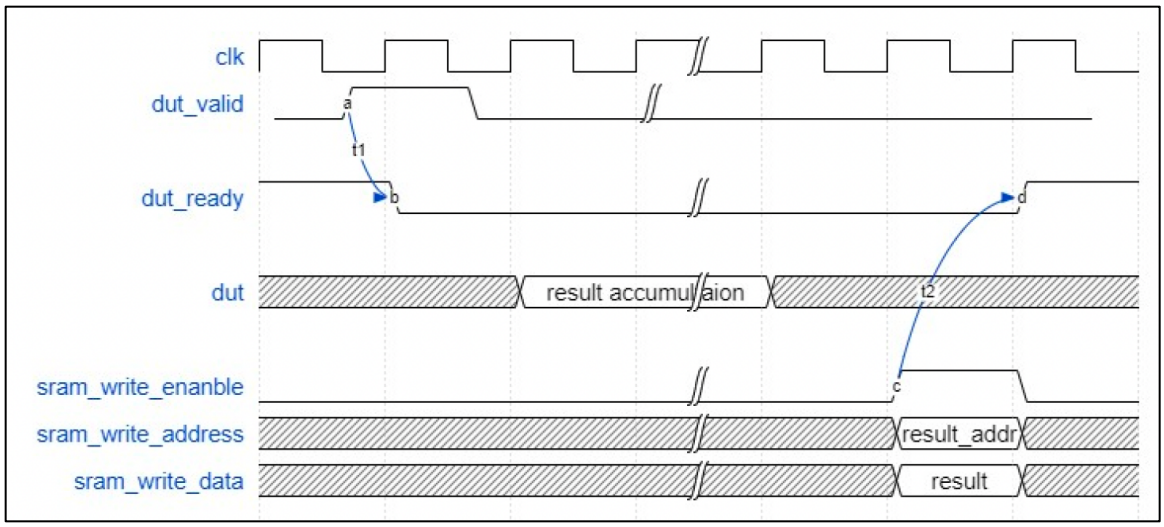
when it finishes the calculation of the first row in the result matrix(More specially, it actually updates when it detects the weight_read_address_next and fetches the last element address in the weight matrix(the desired data actually pops out and is calculated four cycles later, the circuit knows that before performing multiplication by using read_address_next to detect it)

2.3 Detail of innovation

- Ensured the SRAM remains active once the circuit is initiated (dut_valid = 1).
- Implemented a deeply pipelined design with only one stall cycle during result writing to SRAM (MUST).
- Optimized matrix multiplication by performing operations with a transposed matrix without explicitly transposing it.
- Doubled data fetching speed at steps 4 and 5 by utilizing the scratchpad SRAM.
- Designed a flexible circuit capable of supporting matrix multiplication with variable dimensions.

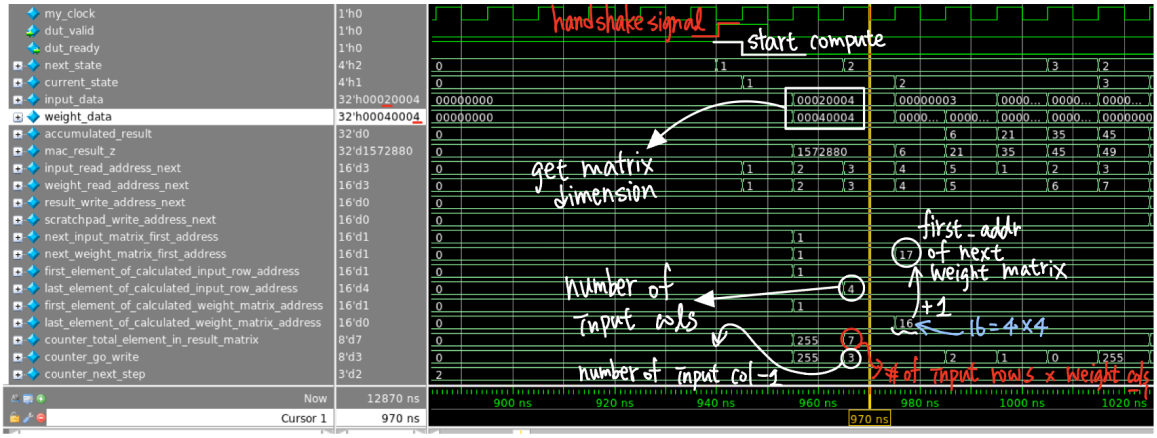
3. Interface Specification

Signal	Width	Function
reset_n	1	Active-low reset signal for the design.
clk	1	System clock signal.
dut_valid	1	handshake signal between testbench and dut,Singals that a valid input can be computed from SRAM
dut_ready	1	Singals that the dut is ready to receive new input from SRAM
dut__tb__sram_input_write_enable	1	Enables writing data to the input SRAM.
dut__tb__sram_input_write_address	16	Specifies the address to write data to in the input SRAM.
dut__tb__sram_input_write_data	32	Data to be written into the input SRAM.
dut__tb__sram_input_read_address	16	Specifies the address to read data from in the input SRAM.
tb__dut__sram_input_read_data	32	Data read from the input SRAM.
dut__tb__sram_weight_write_enable	1	Enables writing data to the weight SRAM.
dut__tb__sram_weight_write_address	16	Specifies the address to write data to in the weight SRAM.
dut__tb__sram_weight_write_data	32	Data to be written into the weight SRAM.
dut__tb__sram_weight_read_address	16	Specifies the address to read data from in the weight SRAM.
tb__dut__sram_weight_read_data	32	Data read from the weight SRAM.
dut__tb__sram_result_write_enable	1	Enables writing data to the result SRAM.
dut__tb__sram_result_write_address	16	Specifies the address to write data to in the result SRAM.
dut__tb__sram_result_write_data	32	Data to be written into the result SRAM.
dut__tb__sram_result_read_address	16	Specifies the address to read data from in the result SRAM.
tb__dut__sram_result_read_data	32	Data read from the result SRAM.
dut__tb__sram_scratchpad_write_enable	1	Enables writing data to the scratchpad SRAM.
dut__tb__sram_scratchpad_write_address	16	Specifies the address to write data to in the scratchpad SRAM.
dut__tb__sram_scratchpad_write_data	32	Data to be written into the scratchpad SRAM.
dut__tb__sram_scratchpad_read_address	16	Specifies the address to read data from in the scratchpad SRAM.
tb__dut__sram_scratchpad_read_data	32	Data read from the scratchpad SRAM.



DUT and testbench handshake behavior

4. Technical Implementation

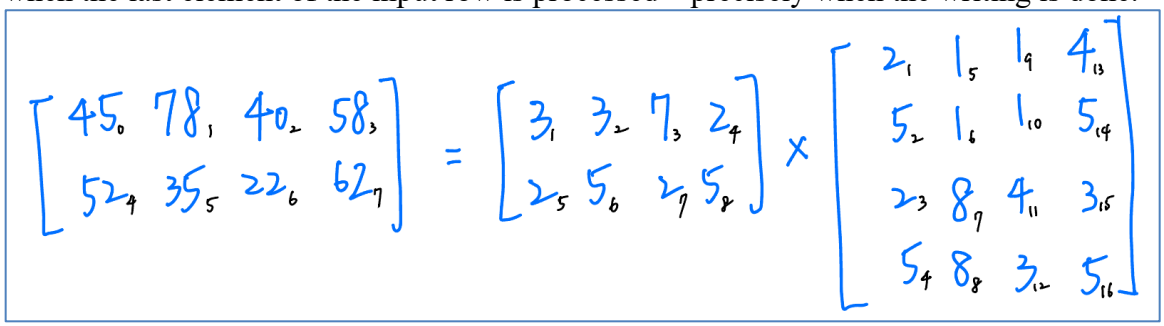


In this timing diagram, before 940 ns, the circuit is in an idle state, waiting for dut_valid = 1 to activate the circuit. During this time, the dut_ready output indicates that the circuit is prepared to begin computations. At 940 ns, the dut_valid signal activates the circuit, and in the next cycle, the state transitions to get_dimension_state (S1).

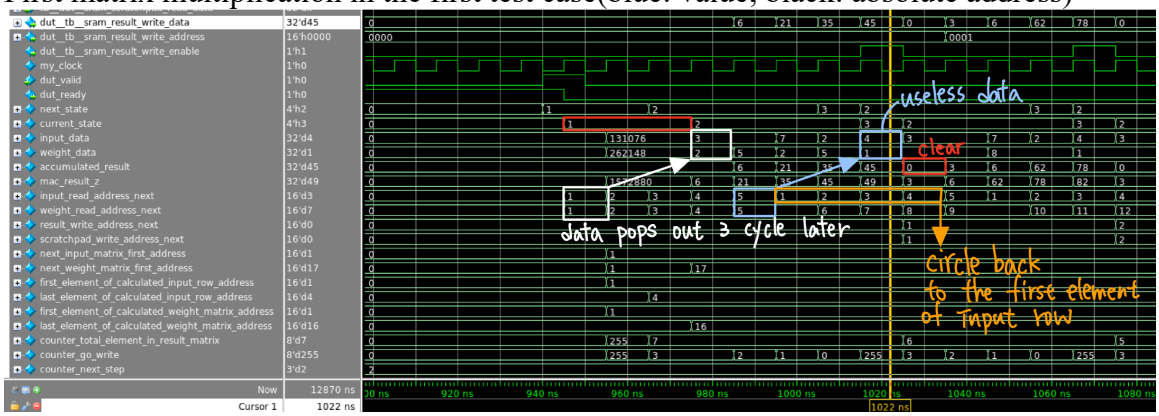
In the S1 state, the circuit reads the matrix dimensions by fetching the first data entry from the SRAM, which encodes the number of rows and columns in a specific format, as shown in the white square box. The first four hexadecimal digits represent the number of rows, while the next four digits indicate the number of columns. For instance, in the input_data signal, the value 00020004 in this timing diagram signifies that the input matrix has two rows (0002) and four columns (0004).

Simultaneously, in the S1 state, the counter_go_write signal is set up for the next state. This counter is configured to number_of_columns - 1 and ensures the circuit correctly times the writing of the accumulated_result to the result SRAM. This setup aligns with

the nature of matrix multiplication, where the accumulated multiplication completes when the last element of the input row is processed—precisely when the writing is done.



First matrix multiplication in the first test case(blue: value; black: absolute address)



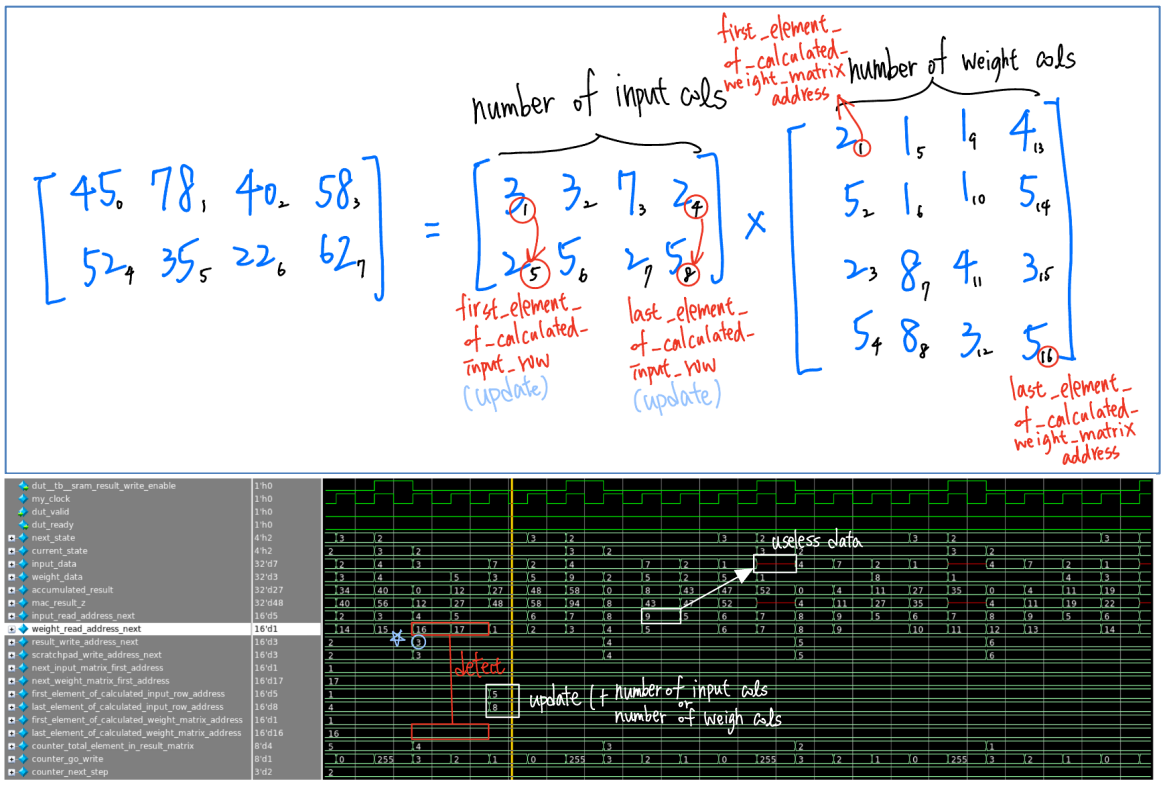
The reason we remain in the S1 state (get_dimension_state) for three cycles is due to the architecture of the DUT and SRAM. It takes three cycles to calculate the next read address, send it to the SRAM, and retrieve the desired data. This ensures that the required data is ready precisely when transitioning to the S2 state (accumulating_state).

A similar concept applies to the S5 through S8 states, where we configure the address and counter to perform the next matrix multiplication state. This timing ensures smooth data flow and proper sequencing for the operations.

I intentionally fetched one incorrect (or unused) data point, which appears during the writing_state (S3), as shown in the blue box above. This is necessary because we clear the accumulated_result after the writing_state. Specifically, we don't want the correct value to appear during the writing_state, as no calculations are performed in this cycle.

This design also prevents potential errors in steps 4 and 5, where both reading from and writing to the result SRAM occur simultaneously. By ensuring the correct value doesn't interfere during the write cycle, we maintain the integrity of the operations.

As shown in the timing diagram, the state transitions from S2 to S3 (writing_state) when the counter_go_write reaches 0. At this point, we effectively loop back to the beginning of the input row, as highlighted in the orange box. The input_read_address_next is updated with the value stored in the first_element_of_calculated_row register, which was recorded during the idle state.



read_address_next transition illustration

When we are about to finish writing the first row of the result matrix (e.g., result_write_address = 3), the input_read_address_next must update to the first element of the next row (e.g., address 5), and weight_read_address_next needs to reset to its first element. We detect this condition by checking whether weight_read_address_next has reached the address of the last element in the weight matrix plus one. If this condition is met, we update both input&weight _read_address_next, as well as the first&last_element_of_calculated_input_row_address registers, to prepare for circling back, as shown in the white box in the figure above.

The reason for detecting when weight_read_address_next reaches the element's address + 1, rather than just the element's address, is to intentionally fetch one unused data point, which will appear during the writing_state(as mentioned in the previous page)

In the timing diagram, instead of circling back to address 1, input_read_address_next updates to address 6, the starting address of the next row, to begin matrix multiplication for the next row in the result SRAM. Meanwhile, weight_read_address_next resets to the first position of the weight matrix.

Notice: Change to the next row takes priority over circle back to the first element of the calculated input row address, that is why this design uses the "if else" statement to declare the priority.

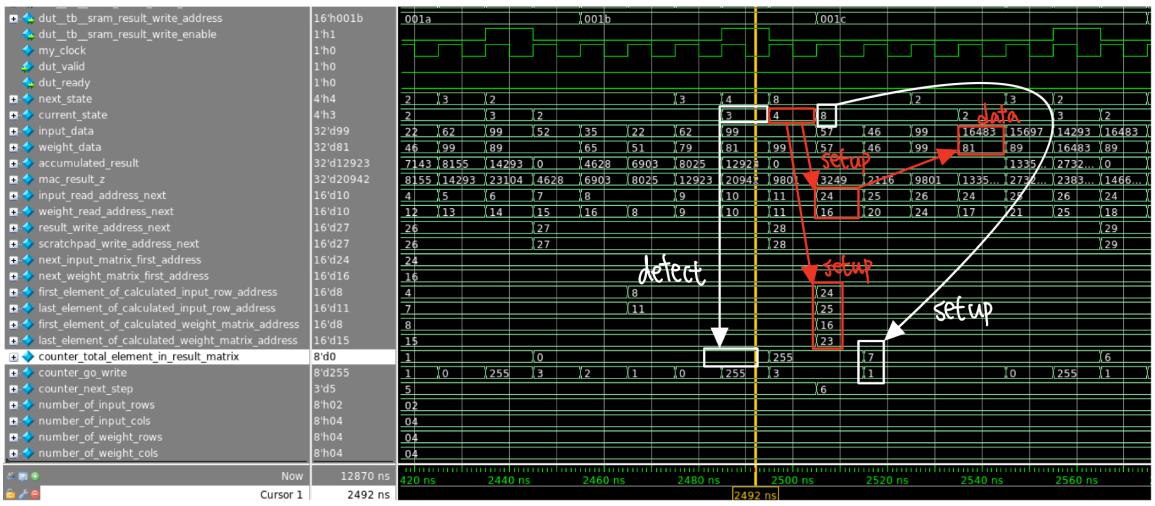


Figure that shows the timing of finishing step 4, S = Q * K^T (Transpose)

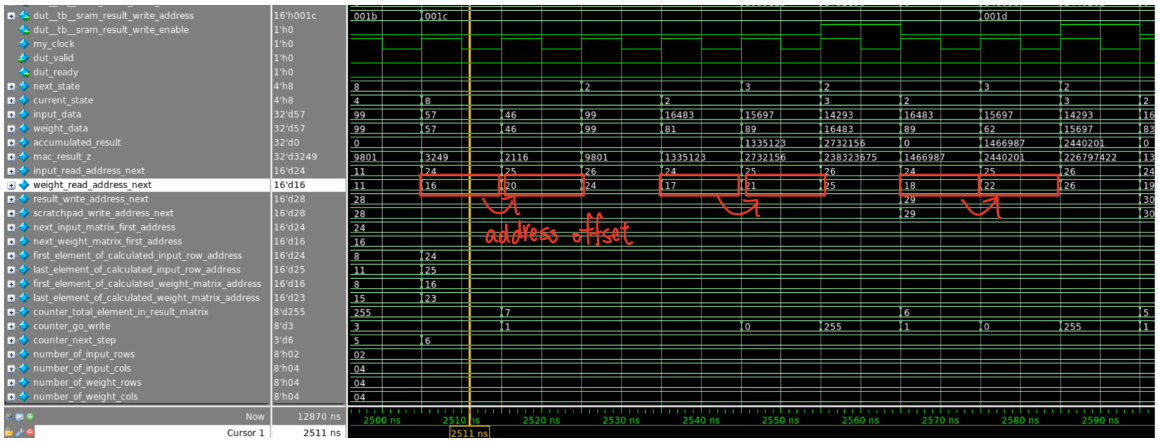
This design detects the completion of one matrix multiplication during the writing_state when counter_total_element_in_result_matrix reaches 0. This counter decreases during the writing_state, and when it hits zero, it indicates that the circuit is going to process the final element of the result matrix. Upon detecting this condition, the current state transitions to next_multiplication_setup_state (S4), where the following addresses are set:

- input_read_address_next and weight_read_address_next
- first_element_of_calculated_input_row_address and
- last_element_of_calculated_input_row_address
- first_element_of_calculated_weight_matrix_address
- last_element_of_calculated_weight_matrix_address

The starting addresses for the next input and weight matrices (not shown in this figure as it pertains to the subsequent step).
Using the counter_next_step, the next_multiplication_setup_state (S4) transitions to the appropriate setup state (setup_multiplication_n_state, where n = 2~5) to initialize the counters for the next matrix multiplication. These counters include:

- counter_total_element_in_result_matrix
- counter_go_write

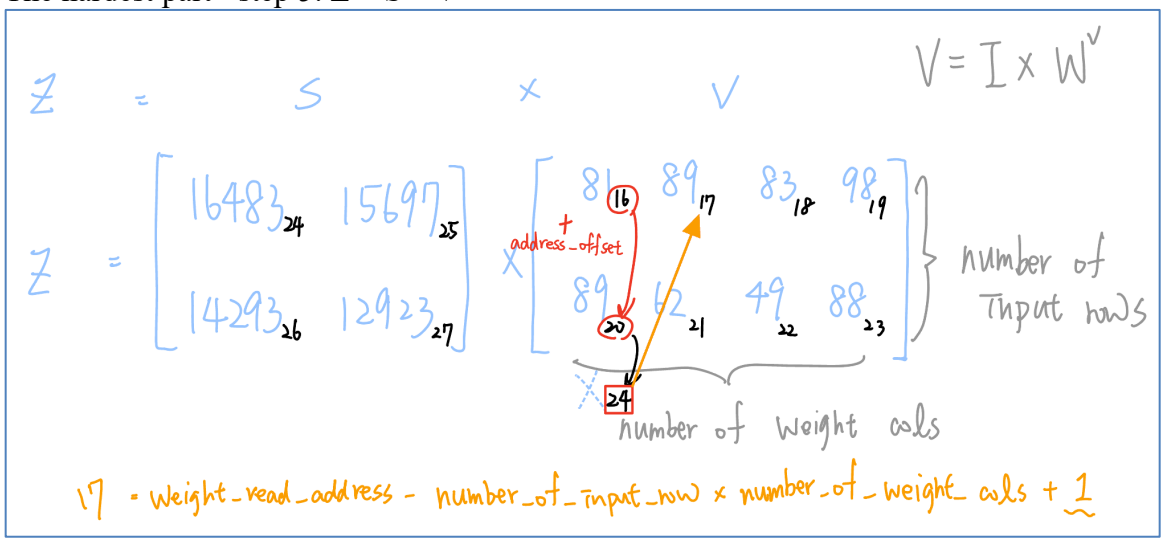
The primary function of the setup_multiplication_*n*_state is to manage the flow of read_address_next to the input and weight SRAM, ensuring that the first desired data will show up when the circuit transitions back to the accumulating_state (S2). This behavior mirrors the role of get_dimension_state (S1) in preparing the initial data for computation.



The hardest part of the entire design is in the last step, where Z = S * V. The challenge includes:

- Perform a matrix multiplication with a transposed matrix without transposing the matrix first to accelerate the speed of the entire system
    - address offset is not simply plus one like the previous four steps
    - change to the next column of the weight matrix without spending extra register(hardware)

The hardest part - step 5: Z = S * V



This figure illustrates the address of V, which looks like a transposition.

To fetch the data from the matrix V in the correct order, we first need to calculate the address offset for the weight_read_address_next. This value is equal to the number of weight columns as $V = I * W^V$, inheriting the dimension property from the matrix W^V.

Second, we move to the next column of the weight matrix by minus the number of input rows * a number of weight cols, then plus one, as shown in the orange part in the above fig.

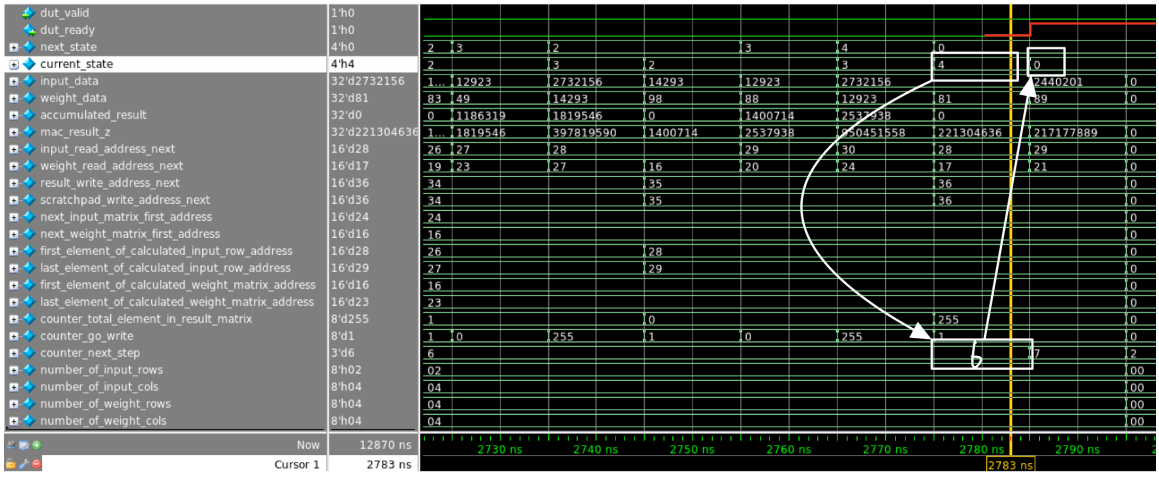
Following the same principle of fetching one unused (or incorrect) data from the SRAM that will show up at writing_state, the system transitions to the next column in the weight matrix when weight_read_addr_next > last_element_calculated_weight_matrix_address + address offset (in this example, address 24).

Since the way of fetching the data in the last step is different from the previous four steps, this design uses the “if else” statement to declare the priority(also use counter_next_step to distinguish the operation of this step). The following code illustrates this:

Partial code:

```
always @(posedge clk or negedge reset_n)begin
  if(!reset_n)
    weight_read_address_next <= 0;
  else if(dut_valid == 1)
    weight_read_address_next <= weight_read_address_next + one_address_offset;
  else if(current_state == idle_state)
    weight_read_address_next <= 0;
  else if(current_state == get_dimension_state)
    weight_read_address_next <= weight_read_address_next + one_address_offset;
  else if((counter_next_step == 6) && (weight_read_address_next == last_element_of_calculated_weight_matrix_address + number_of_weight_cols)
    weight_read_address_next <= first_element_of_calculated_weight_matrix_address;
  else if((counter_next_step == 6) && (weight_read_address_next > last_element_of_calculated_weight_matrix_address) && ((current_state == ac
    weight_read_address_next <= weight_read_address_next - number_of_input_rows*(number_of_weight_cols) + one_address_offset; //change to th
  else if(counter_next_step == 6)
    weight_read_address_next <= weight_read_address_next + number_of_weight_cols; // number_of_weight_cols serve as address offset in transp
  else if((counter_next_step < 6) && (weight_read_address_next == last_element_of_calculated_weight_matrix_address + one_address_offset) &&
    weight_read_address_next <= first_element_of_calculated_weight_matrix_address;
  else if(input_read_address_next == last_element_of_calculated_input_row_address + one_address_offset)
    weight_read_address_next <= weight_read_address_next;
  else if((current_state == next_multiplication_setup_state) && (counter_next_step == 2))
    weight_read_address_next <= next_weight_matrix_first_address;
  else if((current_state == next_multiplication_setup_state) && (counter_next_step == 3))
    weight_read_address_next <= next_weight_matrix_first_address;
  else if((current_state == next_multiplication_setup_state) && (counter_next_step == 4))
    weight_read_address_next <= next_weight_matrix_first_address;
  else if((current_state == next_multiplication_setup_state) && (counter_next_step == 5))
    weight_read_address_next <= next_weight_matrix_first_address;
  else
    weight_read_address_next <= weight_read_address_next + one_address_offset;
end
```

//when counter_next_step == 6, the process is in the last step.



In S4, the circuit also checks whether all calculations are complete. Since the attention equation consists of only five steps, the condition counter_next_step == 6 indicates that all calculations have been performed. As a result, the circuit transitions back to the idle_state in the next cycle, setting dut_ready high to signal that the circuit is ready to compute the next equation, following the handshake protocol.

5. Verification

The testbench instantiates my dut module and SRAMs and sets the dut_valid signal to activate my circuit. After the computation is done, check the correctness by comparing my result in the result SRAM to the answer. Additionally, it also calculates the following:

- Total number of test cases
- Total number of results pass
- Final pass percentage
- Final time result
- Final cycle result

6. Results Achieved

1. Pass all for 4 test cases with 217 result data
2. Total logic (combinational + sequential) area = 9766.72um^2
3. Total cycles for 4 test cases with 217 result data = 1213
4. Total delay for 4 test cases with 217 result data = 6671.5ns
5. Performance 1/(delay x area): 15.2088 ns^{-1}.um^{-2}

```
# INFO: number of testcases:          4
# INFO: DONE WITH RESETING DUT
# INFO:LVL0: ######## Running Test: 1 ########
# INFO: Reading memory file: ../inputs/proj_int_test_self_attn_1_input.dat
# INFO: Reading memory file: ../inputs/proj_int_test_self_attn_1_weight.dat
# INFO: reading ../inputs/proj_int_test_self_attn_1_result.dat
# INFO: Dimension of result matrix [          2,           4]
# ---------------- Checking the Query matrix result------------------
# [          0] Result MATCH: expected_result =         45, dut_result =         45
# [          1] Result MATCH: expected_result =         78, dut_result =         78
# [          2] Result MATCH: expected_result =         40, dut_result =         40
# [          3] Result MATCH: expected_result =         58, dut_result =         58
# [          4] Result MATCH: expected_result =         52, dut_result =         52
# [          5] Result MATCH: expected_result =         35, dut_result =         35
# [          6] Result MATCH: expected_result =         22, dut_result =         22
# [          7] Result MATCH: expected_result =         62, dut_result =         62
# ---------------- Checking the Key matrix result------------------
# [          8] Result MATCH: expected_result =         99, dut_result =         99
# [          9] Result MATCH: expected_result =         57, dut_result =         57
                                .
                                .
                                .
# [         22] Result MATCH: expected_result =         83, dut_result =         83
# [         23] Result MATCH: expected_result =        106, dut_result =        106
# ---------------- Checking the Score matrix result------------------
# [         24] Result MATCH: expected_result =      37356, dut_result =      37356
# ---------------- Checking the Attention matrix result------------------
# [         25] Result MATCH: expected_result =    2502852, dut_result =    2502852
# [         26] Result MATCH: expected_result =    2951124, dut_result =    2951124
# [         27] Result MATCH: expected_result =    3025836, dut_result =    3025836
# [         28] Result MATCH: expected_result =    2801700, dut_result =    2801700
# [         29] Result MATCH: expected_result =    1606308, dut_result =    1606308
# [         30] Result MATCH: expected_result =    1643664, dut_result =    1643664
# [         31] Result MATCH: expected_result =    3100548, dut_result =    3100548
# [         32] Result MATCH: expected_result =    3959736, dut_result =    3959736
# ------------------------------------------------------------------------
# INFO: Total number of cases       : 4
# INFO: Total number of result pass : 217 / 217
# INFO: Final pass percentage       : 100.00
# INFO: Final Time Result           : 12130 ns (period:10ns)
# INFO: Final Cycle Result          : 1213 cycles
```

7. Conclusions

This design introduces a deeply pipelined architecture to perform five-matrix multiplications for the attention mechanism in a transformer, enabling simultaneous data fetching from SRAMs and computation. It features a dedicated sub-module for accumulated matrix multiplications, an efficient datapath for dynamically calculating read addresses, and a 9-state Moore machine to manage data flow and control operations. The design incorporates four counters for timing and state regulation, along with registers for managing matrix dimensions, read/write addresses, and calculated matrix addresses. Its innovative pipelining strategy minimizes stalling, with only a single-cycle delay during write operations. The design successfully passed all four test cases, producing 217 results with a total logic area of 9766.72 μm^2, completing execution in 1213 cycles, and achieving a total delay of 6671.5 ns, resulting in a performance efficiency of 15.2088 $ns^{-1} \cdot \mu m^{-2}$.

Appendix: Address allocation

SRAM input: Address	SRAM input: Content [31:0]
12'h00	[31:16] – Number of matrix A rows, [15:0] – Number of matrix A columns
12'h01	I_1
12'h02	I_2
.	.
.	.
12'h40	I_{64}

SRAM weight: Address	SRAM weight: Content [31:0]
12'h00	[31:16] – Number of matrix B rows, [15:0] – Number of matrix B columns
12'h01	wq_1
12'h02	wq_2
.	.
.	.
12'h100	wq_{256}
12'h101	wk_1
12'h102	wk_2
.	.
.	.
12'h200	wk_{256}
12'h201	wv_1
12'h202	wv_2
.	.
.	.
12'h300	wv_{256}

SRAM Result: Address	SRAM Result: Content [31:0]
12'h00	Q_1
12'h01	Q_2
.	.
.	.
12'h3F	Q_{64}
12'h40	K_1
12'h41	K_2
.	.
.	.
12'h7F	K_{64}
12'h80	V_1
12'h81	V_2
.	.
.	.
12'hBF	V_{64}
12'hC0	S_1
12'hC1	S_2
.	.
.	.
12'hCF	S_{16}
12'hD0	Z_1
12'hD1	Z_2
.	.
.	.
12'h10F	Z_{64}