Final Project: BertEncoder Accelerator Engine

Submission Due Dates:
2024/6/20 23:59

Objective

In the final project, you will develop an accelerator for processing attention and feedforward layers of a quantized BERT model in Verilog. The overall accelerator structure, including data fetching from SRAM, clock handling, and reset mechanisms, is very similar to the HW4 implementation, but may have some differences in the specific details and requirements.

Model Overview

Our target model is based on the pre-trained BERT-Tiny architecture from Hugging Face. We fine-tuned this model on the GLUE benchmark's SST-2 dataset, which focuses on sentiment analysis. This means our model can effectively classify sentences as positive or negative.

Model Architecture

The model architecture is shown in Figure 1 for your reference. You can refer to the original BERT paper for a comprehensive understanding of the BERT model. In this assignment, you will focus on implementing one BertEncoder layer in Figure 1. Figures 2 and 3 provide a more detailed picture of one BertEncoder layer. We have trained a quantized BERT model with 8-bit parameters (e.g., weights, input activations, and output activations) and prepared the test data for you.

```
==========================================================================
Layer (type:depth-idx)                                   Output Shape
==========================================================================
BertForSequenceClassification                            [1, 2]
├─BertModel: 1-1                                         [1, 128]
│    └─BertEmbeddings: 2-1                               [1, 6, 128]
│    │    └─Embedding: 3-1                               [1, 6, 128]
│    │    └─Embedding: 3-2                               [1, 6, 128]
│    │    └─Embedding: 3-3                               [1, 6, 128]
│    │    └─LayerNorm: 3-4                               [1, 6, 128]
│    │    └─Dropout: 3-5                                 [1, 6, 128]
│    └─BertEncoder: 2-2                                  [1, 6, 128]
│    │    └─ModuleList: 3-6                              --
│    └─BertPooler: 2-3                                   [1, 128]
│    │    └─Linear: 3-7                                  [1, 128]
│    │    └─Tanh: 3-8                                    [1, 128]
├─Dropout: 1-2                                           [1, 128]
├─Linear: 1-3                                            [1, 2]
==========================================================================
```

Figure 1: The visualization of the model.

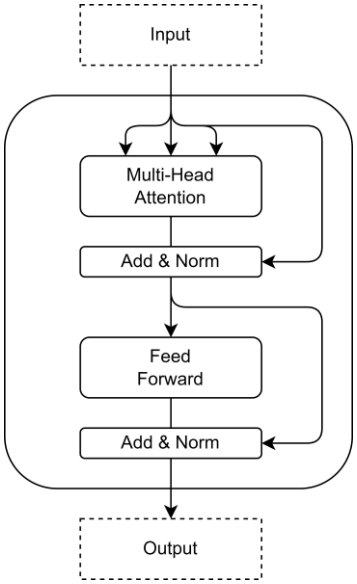

Figure 2: The overall data flow within one BertEncoder.

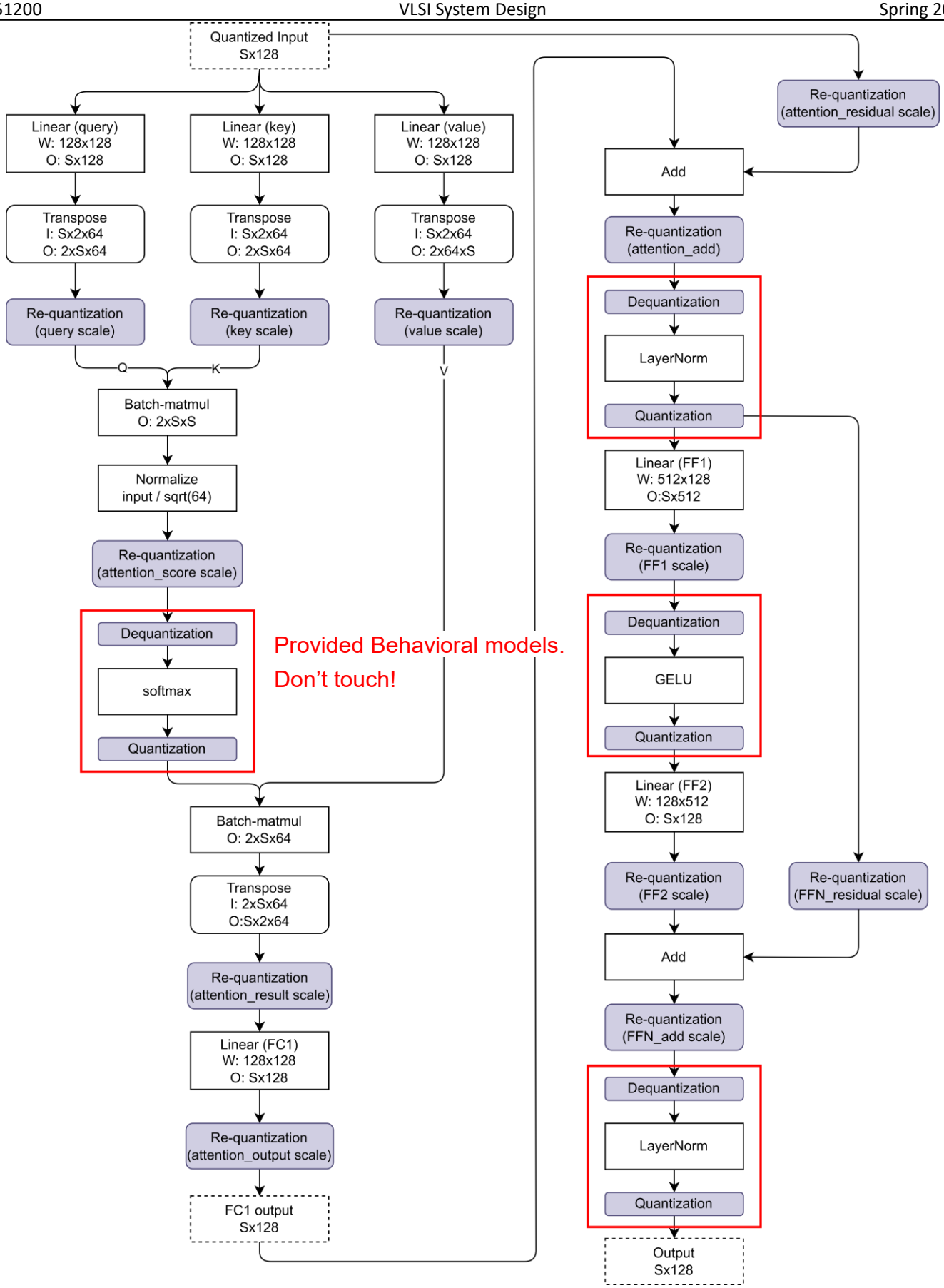

Figure 3: The internal components and their connections within one BertEncoder. The letters W, O, and I provide information about the dimensions and shapes of the weights, outputs, and inputs, respectively. Underneath the re-quantization boxes, the names inside

the parentheses indicate the specific scale used for that re-quantization operation. In each linear operation, the names inside the parentheses indicate the specific weight and bias names required for that operation (Figure 5, Table A.1). The dashed boxes (Quantized input, FC1 output, Output) represent the activation data stored in the SRAM, as shown in Figure 6.

Multi-head attention

In Multi-head attention, the input is donated as $X \in R^{s \times d_{model}}$, where s is the sequence length, and the model dimension $d_{model} = 128$. The sequence length s can be up to 32 in this implementation. The model dimension, d_{model}, represents the length of the vector that each word or token is converted into. In other words, each token in the input sequence is transformed into a vector of length d_{model}. The output size is the same as the input. In this work, we employ head number $h = 2$, and the size of each head is $d = d_{model}/h = 64$. To help you better understand the concept of multi-head attention, we have provided a visualization in Figure 4.

$MultiHead(X) = Concat(head_1, \ldots, head_h)W^{FC1} + bias^{FC1}$, where $W^{FC1} \in R^{hd \times d_{model}}$.

$$head_i = Attention(Query_i, Key_i, Value_i).$$

$$Query_i = XW_i^Q + bias_i^Q, \text{where } W_i^Q \in R^{d_{model} \times d}.$$

$$Key_i = XW_i^K + bias_i^K, \text{where } W_i^K \in R^{d_{model} \times d}.$$

$$Value_i = XW_i^V + bias_i^V, \text{where } W_i^V \in R^{d_{model} \times d}.$$

$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d}}\right)V$, where $softmax$ is applied row-wise.

Transpose

In the diagram, there are also transpose operations applied to the query, key, and value tensors. The original Python code for these transpose operations is as follows:

```
query = query.transpose(1, 0, 2)
key = key.transpose(1, 0, 2)
value = value.transpose(1, 2, 0)
```

The transpose function in NumPy allows you to permute the dimensions of a tensor. In the first two cases, the transpose operation is used to rearrange the dimensions of the query and key tensors from (sequence_length, head_number, head_size) to (head_number, sequence_length, head_size). This is done to align the dimensions for the subsequent matrix multiplications.

Add & Norm

We employ a residual connection around each of the two sub-layers (i.e., multi-head self-attention and feed-forward layer), followed by layer normalization. That is, the output of each sub-layer is

$$LayerNorm(x + Sublayer(x)),$$

where $Sublayer(x)$ is the Multi-head attention layer or Feed Forward layer in Figure 2.

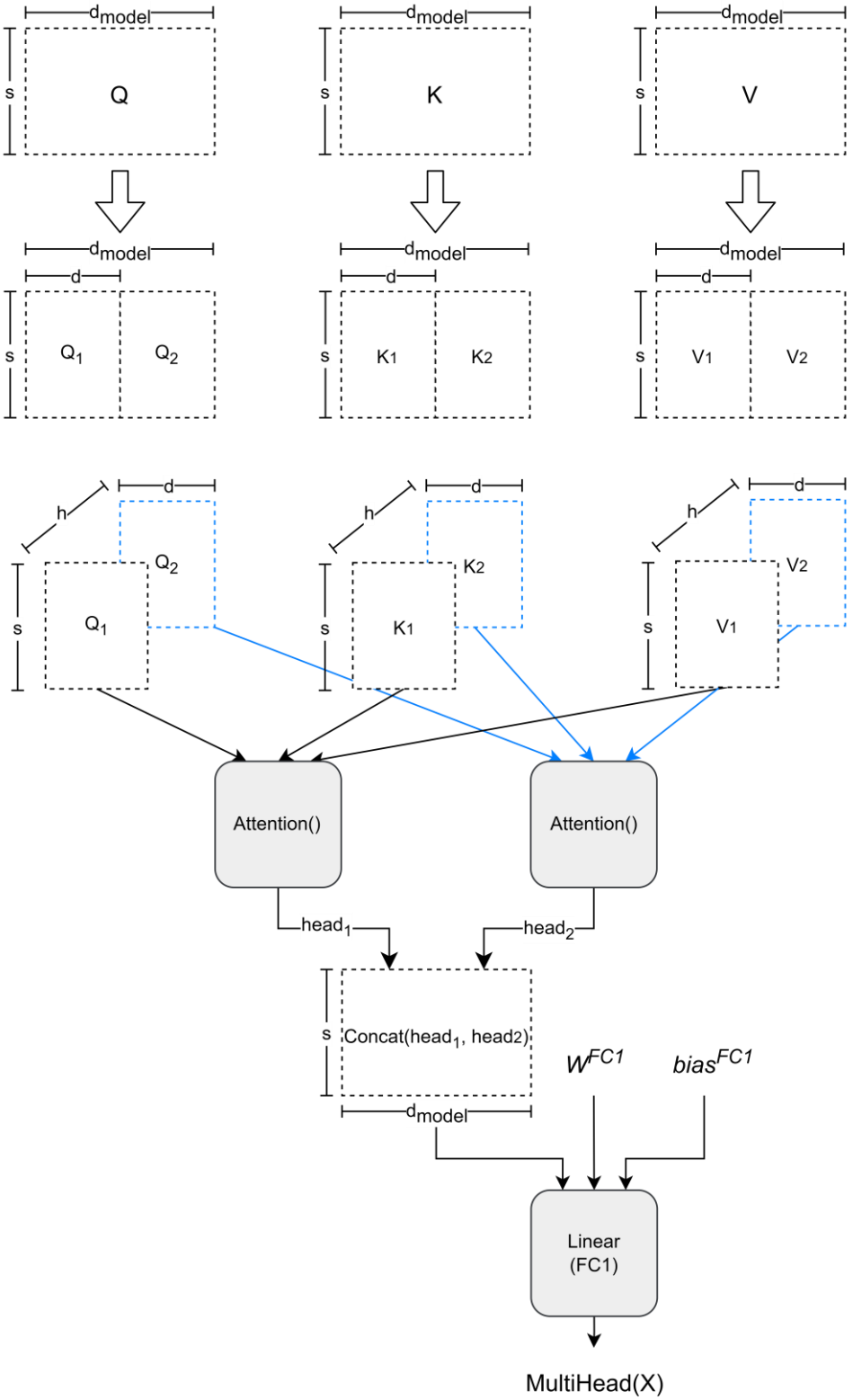

Figure 4: The visualization of the multi-head attention.

Feed Forward

$FFN(X) = GELU(XW_1 + bias_1)W_2 + bias_2$, where $W_1 \in R^{d_{model} \times d_{ff}}$, $W_2 \in R^{d_{ff} \times d_{model}}$. The dimensionality of input and output is $d_{model} = 128$, and the inner-layer has dimensionality $d_{ff} = 512$.

GELU, Softmax, LayerNorm,

The quantization algorithms in our previous homework 1 rely on the linear property of the operator. However, GELU, Softmax, and LayerNorm functions do not possess linear properties.

To address this, we will calculate these three functions in their original precision rather than using 8-bit quantization. We have provided the implementations of GELU, Softmax, and LayerNorm as models in **. /sim/non_linear_model** folder. Simply place the corresponding input on the port and use the Valid-Ready handshake protocol on the module we provide, and the module will send back the calculated answer. If you are interested in more detailed calculations, you can refer to the source code. To help you understand how to use these non-linear modules, we have also provided three examples (softmax_tb.v, GELU_tb.v and layernorm_tb.v) in the **./sim/non-linear_tb** folder.

Please note that some operations in the provided models may not be directly synthesizable, such as the square root function. For those interested in exploring integer-based methods for these three functions, you can refer to the i-BERT paper for more information. However, for the purpose of this homework, please use the provided models as they are and do not modify them.

1 bert_encoder I/O port to the softmax module

$$Softmax(x)_i := \frac{\exp(x_i)}{\sum_{j=1}^{s} \exp(x_j)}, \; where\; x = [x_1, \; \ldots, \; x_s], s = sequence\; length.$$

The input of the softmax module is the result of the matrix multiplication and normalization of Query and Key. For softmax module, each input and data element are 8 bits wide. The in_data input port is 256 bits wide, which means that the softmax module can calculate up to 32 data elements at once (256 bits / 8 bits per data = 32 data elements).

The softmax module supports 1 to 32 sequence length, which is specified by the sequence_length input port. The value of the sequence_length port is set in the testbench file bert_encoder_tb.v.

Please note that you must send a whole row of data (1 × sequence length) to the softmax module. This means that the input data should correspond to a complete row of the matrix multiplication result between Query and Key.

Signals	Width	I/O	Description
softmax_data_in_valid	1	Output	Indicates a new computation request. When set, input data fields must be valid.
softmax_data_in_ready	1	Input	Indicates that the softmax module is ready to accept a new computation request.
softmax_in_data	256	Output	The input data of softmax
sequence_length	8	Input	Sequence length
softmax_in_scale	32	Output	The scale of softmax input data
softmax_out_scale	32	Output	The scale of the softmax output data
softmax_data_out_valid	1	Input	Indicates that the softmax module has valid data as its output
softmax_data_out_ready	1	Output	Indicates that the module downstream is ready to accept the output data
softmax_out_data	256	Input	The result of softmax

2 bert_encoder I/O port to the GELU module

$$GELU(x) \coloneqq x \cdot \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right],\ where\ \operatorname{erf}(x) \coloneqq \frac{2}{\sqrt{\pi}}\int_0^x \exp(-t^2)\,dt.$$

The GELU module expects the input data to be 256 bits wide. Each element in the input data is 8 bits wide, so the module can process up to 32 elements (32 x 8 bits = 256 bits) in a single computation.

You can provide the input data element within 32, but you need to ensure that it is padded with zeros to form a complete 256-bit input. For example, if you want to calculate only one element using the GELU module, you should pad the data with zeros to create a 256-bit input and then send it to the GELU module.

The output data from the GELU module will have the same number of elements as the input data, and each output element will be 8 bits wide.

Signals	Width	I/O	Description
gelu_data_in_valid	1	Output	Indicates a new computation request. When set, input data fields must be valid.
gelu_data_in_ready	1	Input	Indicates that the GELU module is ready to accept a new computation request.

gelu_in_data	256	Output	The input data of GELU
gelu_in_scale	32	Output	The scale of input data
gelu_out_scale	32	Output	The scale of the output data
gelu_data_out_valid	1	Input	Indicates that the GELU module has valid data as its output
gelu_data_out_ready	1	Output	Indicates that the module downstream is ready to accept the output data
gelu_out_data	256	Input	The result of GELU

3 bert_encoder I/O port to the layernorm module

$\tilde{x} = \left(\frac{x-\mu}{\sigma}\right)\gamma + \beta$, where $\mu = \frac{1}{d_{model}}\sum_{i=1}^{d_{model}} x_i$, $\sigma = \sqrt{\frac{1}{d_{model}}\sum_{i=1}^{d_{model}}(x_i - \mu)^2}$, $\gamma = layernorm\ weights, \beta = layernorm\ bias, d_{model} = 128.$

Each element in the input data is 8 bits wide, and in each transmission the module expects 32 elements (32 x 8 bits = 256 bits) to be sent. You must allocate the full 256 bits when sending data to the layernorm module.

The input to the layer of layernorm has a shape of $s \times d_{model}$, where s is the sequence length, $d_{model} = 128$. To start the computation, the layernorm module requires a total of 128 elements in the same row, which means you need to **send data to the layernorm module 4 times**. You can send the data successively (in 4 cycles) or separately, depending on your design.

Once the layernorm module receives the required 128 elements, it will start the computation. The module will output the result as 128 elements (sent as 4 transmissions of 32 elements each), where each output element is 8 bits wide. You can receive the output data successively (in 4 cycles) or separately, just the same as when sending the input data.

Signals	Width	I/O	Description
layernorm_data_in_valid	1	Output	Indicates a new computation request. When set, input data fields must be valid.
layernorm_data_in_ready	1	Input	Indicates that the layernorm module is ready to accept a new computation request.
layernorm_in_data	256	Output	The input activation of layernorm
layernorm_weights	256	Output	The input weight of layernorm
layernorm_bias	256	Output	The input bias of layernorm
layernorm_in_scale	32	Output	The scale of input activation

layernorm_weight_scale	32	Output	The scale of input weight
layernorm_bias_scale	32	Output	The scale of the input bias
layernorm_out_scale	32	Output	The scale of the output data
layernorm_data_out_valid	1	Input	Indicates that the layernorm module has valid data as its output
layernorm_data_out_ready	1	Output	Indicates that the module downstream is ready to accept the output data
layernorm_out_data	256	Input	The result of layernorm

Re-quantization

When you apply re-quantization, make sure to include the following step:

$$Requantization(x) = clamp((x \times scale) \gg 16\ , -128, 127)$$

After applying the scale to x, you need to perform a right shift of 16 bits. This is because we have rounded the scale to the 16th place using a software trick.

Testbench & SRAM

1. A testbench **./sim/bert_encoder_tb.v** for your validation. It can load pattern and weight files into weight SRAM and activation SRAM, respectively, and validate the activation SRAM, where you should store your computation results. This testbench provides a simple result comparison for BertEncoder layer's output activation. You can modify and enhance its debugging capability accordingly. **But note that we will use the original testbench to justify your design.**
   Note: Feel free to modify the default END_CYCLE if you need the larger one. Remember to add it in the report.
2. We have two dual-port SRAMs in **./sim/sram_model**: one is the weight SRAM, the other is the activation SRAM. For the given dual-port SRAM, you can perform two 128-bit memory accesses of separate addresses in the same clock cycle.
3. Figure 5 shows the layout of the weight SRAM. For scales, we store four 32-bit scales in each address. For weights, we store sixteen 8-bit weights in each address. For biases, sixteen 8-bit bias is stored in every address.
4. Figure 6 shows the layout of activation SRAM. The testbench will load the quantized input from offset 0 to offset 255 in SRAM. Each address holds sixteen 8-bit inputs. You should place sixteen 8-bit quantized data in every address for FC1 output from offset 256 to offset 511 in SRAM and the results of the whole BertEncoder layer from offset 512 to offset 767 in SRAM.
5. You are free to use the space marked as "free to use" in Figure 5 and Figure 6 for any purpose. During validation, we will not verify the contents of those memory blocks.

6. You should use the testbench data to validate your design. The testbench data includes the quantized input, weight, bias, quantization scale, and golden data, which have been pre-processed to match the format in Figures 5 and 6. In addition, please check TODO in testbench to replace your own file name and the sequence length.
   Note: Refer to the **Appendix** for further discussion on the "free to use" and data arrangement in SRAMs.
7. TA will use unreleased patterns to evaluate your design.

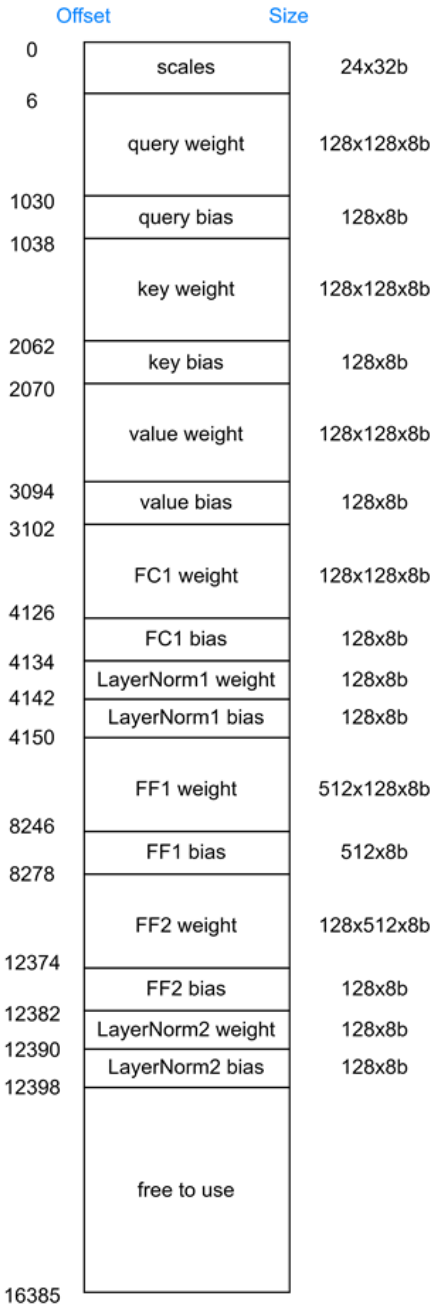

Figure 5: Layout of weight SRAM.

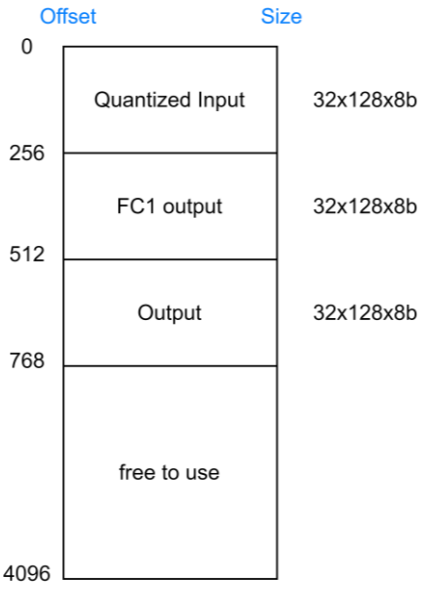

Figure 6: Layout of the activation SRAM

Design

1. A template **./hdl/bert_encoder.v** is provided. **DO NOT** change the I/O declaration.
2. **Input and output delay constraints** are added in the synthesis script. Make sure to add flip-flops after input data ports (**except for the clk, rst_n port, and the I/O of non-linear modules**) and before output data ports.

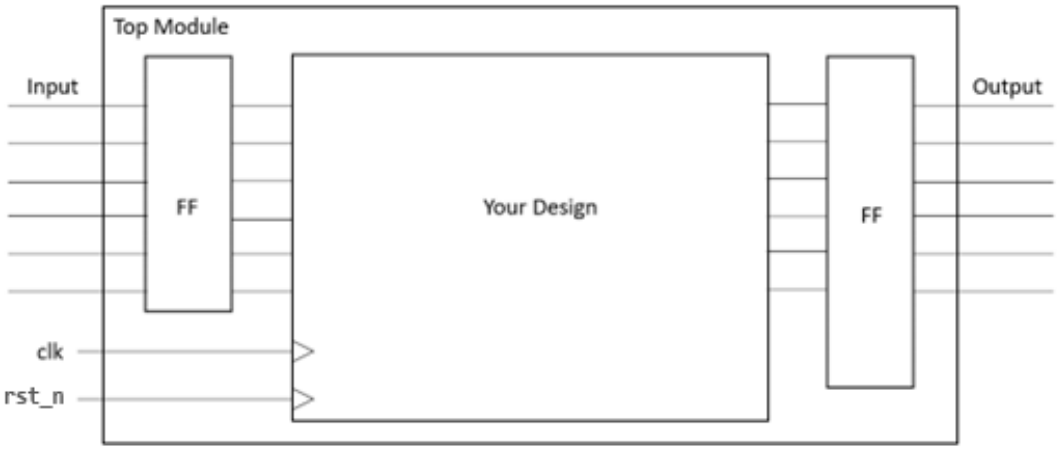

Figure 7: I/O-registered design style.

3. I/O port descriptions

Signals	Width	I/O	Description
clk	1	Input	Clock.
rst_n	1	Input	**Active-low** reset.
compute_start	1	Input	**Single-cycle active-high pulse**. The testbench uses this signal to inform the engine to start computing.
compute_finish	1	Output	You should pull up a **single-cycle active-high pulse** to inform the testbench the computation is finished.
sram_weight_wea0	16	Output	Each bit represents the byte-write enable for SRAM port 0. E.g., 16'b0000000000001001 means RAM[addr][31:24] = wdata[31:24] and RAM[addr][7:0] = wdata[7:0], leaving RAM[addr][23:8] and RAM[addr][127:32] untouched. Please refer to the SRAM behavior code to know how it works.
sram_weight_addr0	16	Output	Read/write address of port 0 in the weight SRAM.

sram_weight_wdata0	128	Output	Write data of port 0 in the weight SRAM.
sram_weight_rdata0	128	Input	Read data of port 0 in the weight SRAM.
sram_weight_wea1	16	Output	Each bit represents the byte-write enable for SRAM port 1.
sram_weight_addr1	16	Output	Read/write address of port 1 in the weight SRAM.
sram_weight_wdata1	128	Output	Write data of port 1 in the weight SRAM.
sram_weight_rdata1	128	Input	Read data of port 1 in the weight SRAM.
sram_act_wea0	16	Output	Each bit represents the byte-write enable for SRAM port 0.
sram_act_addr0	16	Output	Read/write address of port 0 in the activation SRAM.
sram_act_wdata0	128	Output	Write data of port 0 in the activation SRAM.
sram_act_rdata0	128	Input	Read data of port 0 in the activation SRAM.
sram_act_wea1	16	Output	Each bit represents the byte-write enable for SRAM port 1.
sram_act_addr1	16	Output	Read/write address of port 1 in the activation SRAM.
sram_act_wdata1	128	Output	Write data of port 1 in the activation SRAM.
sram_act_rdata1	128	Input	Read data of port 1 in the activation SRAM.

Simulation & Synthesis

For the gate-level simulation, you should modify the clock period in ./sim/ bert_encoder_tb.v to meet the clock constraint. Other information about simulation and synthesis, please refer to the descriptions in Homework 4.

Grading Policy

1. RTL simulation pass (30%)
2. Spyglass report to show your design is synthesizable (10%)
3. Gate-level simulation pass (17%)

Note: Setup/hold time violation is not acceptable, even though you can pass the gate-level simulation.

4. Report (30%)
5. Performance ranking (13%)

   This part is only for those who passed the gate-level simulation.

   Please fill following items to this form: https://reurl.cc/p3aWY4

   (a) Name & Student ID

   (b) **Gate-level simulation** clock period (e.g. 12 ns)

   (c) **Gate-level simulation** latency (e.g. 23729 cycles)

   (d) **Total cell area** of Design Compiler report (e.g. 77649 um^2)

   Note: If you didn't fill out the form, you will not get the grade for this part!

Report

1. Design concept (You may write the report in Chinese):
   - Please provide block diagrams of the overall hardware architecture and each component. Please explain the purpose and the way they work together.
   - Please provide state diagram of your FSM (if any), and its detailed description. If no FSM is used in your design, please explain your control flow.
   - Please explain your dataflow in detail.
   - Please explain the usage of the free-to-use space in the SRAMs.
   - Any additional information you want to provide.

2. Result

Item	Description	Unit
RTL simulation	(PASS / FAIL)	---
Gate-level simulation	(PASS / FAIL)	---
Gate-level simulation clock period	(XXX)	ns
Gate-level simulation latency	(XXX)	cycles
Total cell area	(XXX)	um^2

3. Contribution (skip this part if you are in a 1-person group)

Item	Student1	Student2
Architectural design		
Coding		
Report writing		
… (you are free to adjust or add the items base on your situation)		

4. END_CYCLE (optional)
   - If you need a larger END_CYCLE than the default, please let us know what value you are using.

END_CYCLE	

5. Others (optional)
   - Suggestions or comments about this class to the teacher or TA.

Submission

1. Please submit the following files to EECLASS (or 20-point penalty if not following the rules).
   - bert_encoder.v # Include top module bert_encoder and all other modules. Please integrate all your code into this single file, excluding SRAM_weight_16384x128b.v, SRAM_activation_4096x128b.v, GELU.v, softmax.v and layernorm.v.
   - bert_encoder_syn.v # Gate-level netlist
   - bert_encoder_syn.sdf # Standard Delay Format file for gate-level simulation
   - spyglass.rpt # Please follow the tutorial to generate this file.
   - report_timing.log # Generated by Design Compiler
   - report_area.log # Generated by Design Compiler
   - <student_id>_<student_id>_final_project_report.pdf # Use the template we provide.
2. All work submitted, including both source code and reports, must be completed independently by you and your team member. We enforce a strict policy against academic misconduct, including dishonesty, cheating, and plagiarism. To ensure the integrity of submissions, we will employ a plagiarism detection tool. Submissions with high similarity scores (e.g., above 25%) will be thoroughly reviewed and subject to disciplinary action based on the infraction.
3. **DO NOT** compress submitted files!

Appendix A: SRAM Data Arrangement

Data arrangement in SRAMs

This section demonstrates how the activations and weights are arranged in the activation SRAM and weight SRAM. Generally, the data arrangement is the same as that in HW4. **Each line has a 128-bits HEX data.**

Table A.1: Weight files data comparison

weights.csv			Original weight files	
Line #	Bit position	Data (HEX)	item	Data (DEC)
0	[31:0]	000000d8	query scale	216
	[63:32]	000000bc	key scale	188
	[95:64]	000000e1	value scale	225
	[127:96]	0000003c	attention_score scale	60
1	[31:0]	00004ea4	softmax_in scale	20132
	[63:32]	000001db	softmax_out scale	475
	[95:64]	000002d9	attention_result scale	729
	[127:96]	000001c3	attention_output scale	451
2	[31:0]	0001df72	attention_residual scale	122738
	[63:32]	0000567b	attention_add scale	22139
	[95:64]	000007cb	attention_layernorm_in scale	1995
	[127:96]	000003ab	attention_layernorm_weights scale	939
3	[31:0]	00000415	attention_layernorm_bias scale	1045
	[63:32]	000005db	attention_layernorm_out scale	1499
	[95:64]	000000f6	FF1 scale	246
	[127:96]	000007d5	GELU_in scale	2005
4	[31:0]	00000430	GELU_out scale	1072
	[63:32]	000005cb	FF2 scale	1483
	[95:64]	000338e9	FFN_residual scale	211177
	[127:96]	00005138	FFN_add scale	20792
5	[31:0]	000005ba	FFN_layernorm_in scale	1466
	[63:32]	000002f3	FFN_layernorm_weights scale	755
	[95:64]	0000013c	FFN_layernorm_bias scale	316
	[127:96]	00000539	FFN_layernorm_out scale	1337

6	[7:0]	ed	query weight 0	-19
	[15:8]	fd	query weight 1	-3
	[23:16]	e8	query weight 2	-24
	[31:24]	fa	query weight 3	-6
	[39:32]	0b	query weight 4	11
	[47:40]	09	query weight 5	9
	[55:48]	11	query weight 6	17
	[63:56]	ec	query weight 7	-20
	[71:64]	fa	query weight 8	-6
	[79:72]	d0	query weight 9	-48
	[87:80]	12	query weight 10	18
	[95:88]	11	query weight 11	17
	[103:96]	e6	query weight 12	-26
	[111:104]	15	query weight 13	21
	[119:112]	d7	query weight 14	-41
	[127:120]	f6	query weight 15	-10
7	[7:0]		query weight 16	
	…	…	…	
	…	…	…	
	[127:120]		query weight 31	
…	…	…	…	

Table A.2: Activation files data comparison

golden.csv			Original weight files	
Line #	Bit position	Data (HEX)	item	Data (DEC)
0	[7:0]	04	input 0	4
	[15:8]	00	input 1	0
	[23:16]	d1	input 2	-47
	[31:24]	00	input 3	0
	[39:32]	00	input 4	0
	[47:40]	ff	input 5	-1
	[55:48]	ff	input 6	-1
	[63:56]	04	input 7	4
	[71:64]	04	input 8	4
	[79:72]	fc	input 9	-4
	[87:80]	fe	input 10	-2
	[95:88]	02	input 11	2
	[103:96]	ff	input 12	-1
	[111:104]	06	input 13	6

	[119:112]	fe	input 14	-2
	[127:120]	05	input 15	5
1	[7:0]		input 16	
	…	…	…	
	…	…	…	
	[127:120]		input 31	
…	…	…	…	
256	[7:0]		output 0	
	…	…	…	
	…	…	…	
	[127:120]		output 15	

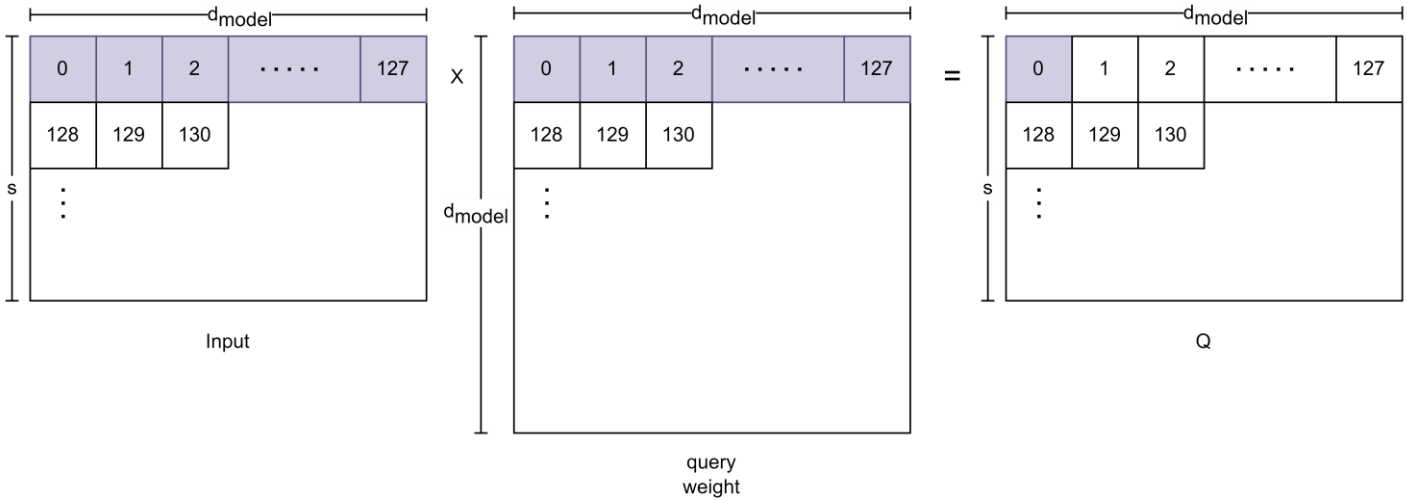

Figure A.1: Visualization of the query matrix multiplication and their position order in SRAM.

"Free to Use" Space in SRAMs

You can store these values in the "free to use" space of the SRAM to validate your intermediate results. Each element in the CSV file is 8 bits wide, so each address in the SRAM can store sixteen elements.

You may store additional intermediate results of your hardware such as the outputs of query, key, or batch-matmul operations, in the "free to use" space during the operation. You must detail the usage explicitly in the report. (Please note that any misuse is prohibited.)

You have the flexibility to decide how many elements to store in each address and how many bits each element should occupy. The only requirement is to ensure that the calculated values are correct.

Please note that we will not verify the contents of the "free to use" space during

the validation process. We will only validate the output results stored in the locations specified in Figure 6.

Provided pattern

We have provided two sets of pattern files for your reference and validation. The filenames are distinguished by appending "0" or "1" at the end. The sequence length for pattern 0 is 28, while the sequence length for pattern 1 is 32. Please check TODO in testbench to set the correct sequence length according to the pattern you are using. In the following description, we will refer to the filenames without the "0" or "1" suffix, using the general filename.

input.csv contains the quantized input data, which should be placed in the "Quantized Input" section of the SRAM, as shown in Figure 6.

golden.csv contains the expected output data, including the quantized input, FC1 output, and final output in Figure 6.

To help you verify the correctness of each function implementation, we have also provided intermediate output files for each function. You can choose to store these intermediate outputs in the "free to use" space of the SRAM for comparison. The correspondence between the intermediate output filenames and their respective functions will be indicated in red text in Figure A.2.

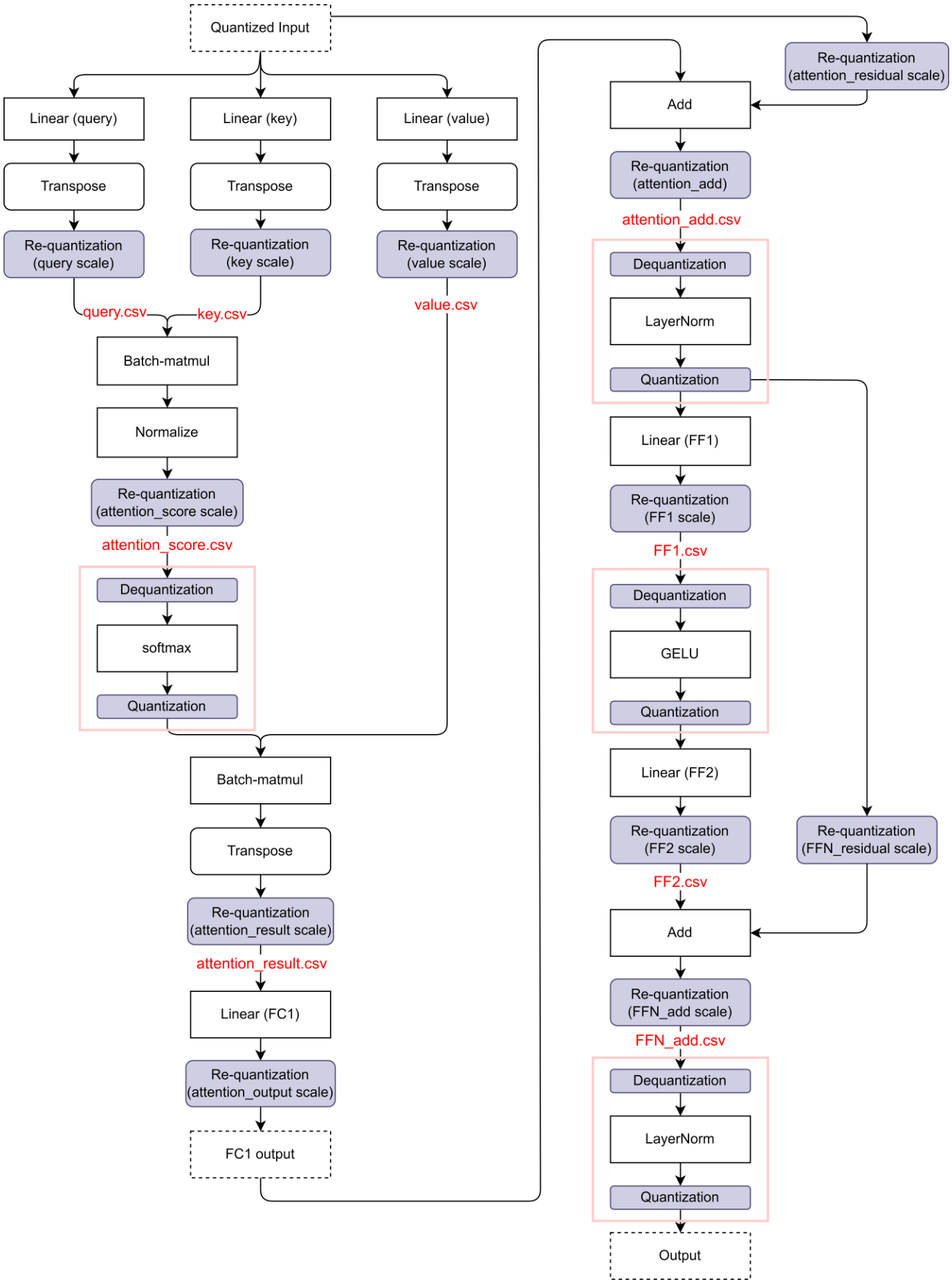

Figure A.2: Mapping of intermediate output filenames to the corresponding functions.