Homework 2: LeNet Accelerator Engine part1

Submission Due Dates:
2025/3/30 23:59

Objective

In this homework, you will implement a part of the CNN accelerator for the quantized LeNet model, specifically focusing on the fully connected layer. The remaining layers will need to be implemented in the next homework (HW3).

Model Architecture

In Homework 1, we trained a quantized LeNet model with 8-bit parameters (such as filter weights, input activations, and output activations). In this homework, you will need to implement a portion of the accelerator engine to compute the first fully connected layer of LeNet in Verilog. The CNN model is provided for your reference, as in Homework 1:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Net                                      --                        --
├─Sequential: 1-1                        [1, 6, 28, 28]            --
│    └─Conv2d: 2-1                       [1, 6, 28, 28]            150
│    └─ReLU: 2-2                         [1, 6, 28, 28]            --
├─Sequential: 1-2                        [1, 6, 14, 14]            --
│    └─MaxPool2d: 2-3                    [1, 6, 14, 14]            --
├─Sequential: 1-3                        [1, 16, 10, 10]           --
│    └─Conv2d: 2-4                       [1, 16, 10, 10]           2,400
│    └─ReLU: 2-5                         [1, 16, 10, 10]           --
├─Sequential: 1-4                        [1, 16, 5, 5]             --
│    └─MaxPool2d: 2-6                    [1, 16, 5, 5]             --
├─Sequential: 1-5                        [1, 120, 1, 1]            --
│    └─Conv2d: 2-7                       [1, 120, 1, 1]            48,000
│    └─ReLU: 2-8                         [1, 120, 1, 1]            --
├─Sequential: 1-6                        [1, 84]                   --
│    └─Linear: 2-9                       [1, 84]                   10,080
│    └─ReLU: 2-10                        [1, 84]                   --
├─Sequential: 1-7                        [1, 10]                   --
│    └─Linear: 2-11                      [1, 10]                   840
==========================================================================================
```

Testbench & SRAM

1. There is a testbench located at ./sim/lenet_tb.v for validation. It can load pattern and weight files into the weight SRAM and the activation SRAM, respectively. It can also validate the content in the activation SRAM, where you should store your computation results. This testbench provides a straightforward method to compare the FC1 layer's activations. You can modify and enhance its debugging capabilities as needed. However, please note that we will use the original testbench to validate **your design.**

   Note: If you need a larger END_CYCLE, feel free to adjust the default one. Remember to include the explanation in the report.
2. We have two dual-port SRAMs in **./sim/sram_model**: one is the weight SRAM, and the other is the activation SRAM. For the given dual-port SRAM, you can perform two 32-bit memory accesses of separate addresses in the same clock cycle.
3. Figure 1 shows the layout of the weight SRAM. In the Conv1 and Conv2 weight layers, we store five 8-bit weights for every two addresses (40 bits for the weights and 24 bits for blank data). It is possible to access five weights in one clock cycle. Note that the Conv3 layer can be treated as a fully-connected layer. For the Conv3, FC1, and FC2 weight layers, we store four 8-bit weights at each address. For FC2 biases, one 32-bit bias is stored at every address. **In this homework, you only need the FC1's weights, which are located from offset 13020 to offset 15539.**

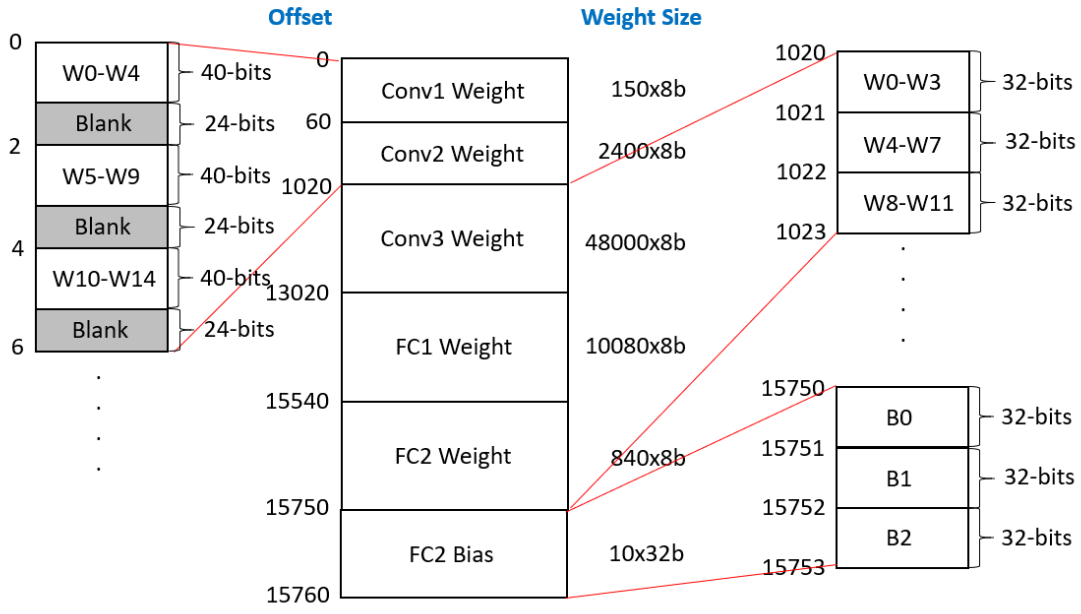

Figure 1: Layout of weight SRAM.

4. Figure 2 shows the layout of the activation SRAM. Initially, all SRAM locations, except offsets 692 to 721, will be zero. The testbench will load the Conv3 output activations

from **offset 692 to offset 721** in SRAM. Each address holds four 8-bit inputs. For the FC1 layer, you should also place four 8-bit quantized data at every address for the next fully connected computation. That means you need to place the output activations of the FC1 at offsets **722-742**.

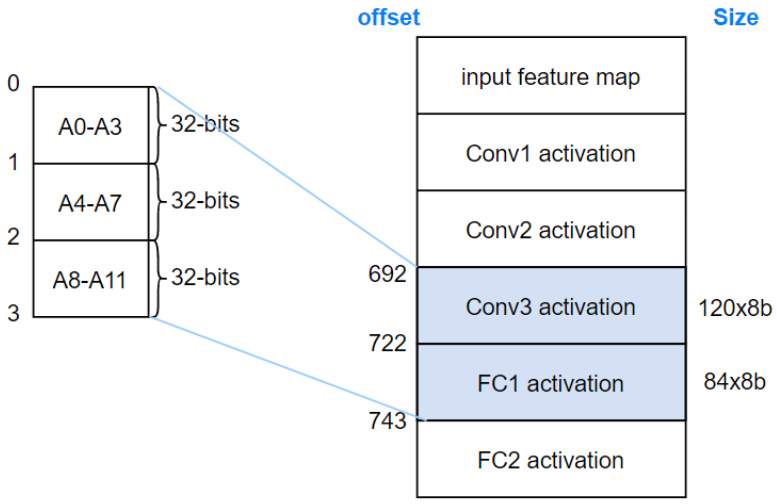

Figure 2: Layout of the activation SRAM.

5. We provide golden data for you to validate your design, which has been preprocessed to match Figures 1 and 2. We recommend that you trace and understand the testbench code, as you will need to do the data preprocessing yourself in the upcoming Homework 3.
   Note 1: The quantization scale factor has already been included in the homework, so don't worry about it.
   Note 2: Refer to the **Appendix** for further discussion on the data arrangement in SRAMs.
6. TA will use unreleased patterns to evaluate your homework.

Design

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

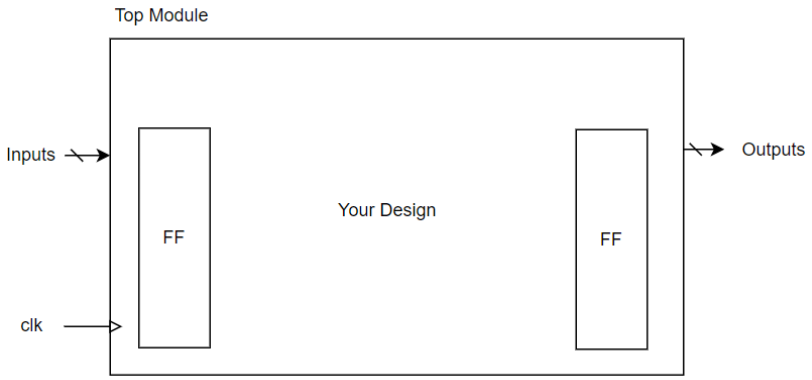

Figure 3: I/O-registered design style.

3. I/O port descriptions:
   In order to allow everyone to continue working on HW3 based on the foundation of HW2, our IO template is based on HW3. Therefore, some of the I/Os that are used in this homework, such as the scale_CONV1, scale_CONV2, etc., can be ignored for this homework. However, please do not modify the I/Os even though they are not used in this homework.

Signals	**Width**	**I/O**	**Description**
clk	1	Input	**Positive-edge-triggered** 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.
scale_CONV1	32	Input	Quantization scale factor for CONV1
scale_CONV2	32	Input	Quantization scale factor for CONV2
scale_CONV3	32	Input	Quantization scale factor for CONV3
scale_FC1	32	Input	Quantization scale factor for FC1
scale_FC2	32	Input	Quantization scale factor for FC2
sram_weight_wea0	4	Output	Each bit represents the byte-write enable for SRAM port 0. E.g., 4'b1001 means RAM[addr][31:24] = wdata[31:24] and RAM[addr][7:0] = wdata[7:0], leaving RAM[addr][23:8] untouched. Please refer to the SRAM behavior code to know how it works.
sram_weight_addr0	16	Output	Read/write address of the port 0 in the weight SRAM.
sram_weight_wdata0	32	Output	Write data of the port 0 in the weight SRAM.
sram_weight_rdata0	32	Input	Read data of the port 0 in the weight SRAM.
sram_weight_wea1	4	Output	Each bit represents the byte-write enable for the

			SRAM port 1.
sram_weight_addr1	16	Output	Read/write address of the port 1 in the weight SRAM.
sram_weight_wdata1	32	Output	Write data of the port 1 in the weight SRAM.
sram_weight_rdata1	32	Input	Read data of the port 1 in the weight SRAM.
sram_act_wea0	4	Output	Each bit represents the byte-write enable for the SRAM port 0.
sram_act_addr0	16	Output	Read/write address of the port 0 in the activation SRAM.
sram_act_wdata0	32	Output	Write data of the port 0 in the activation SRAM.
sram_act_rdata0	32	Input	Read data of the port 0 in the activation SRAM.
sram_act_wea1	4	Output	Each bit represents the byte-write enable for the SRAM port 1.
sram_act_addr1	16	Output	Read/write address of the port 1 in the activation SRAM.
sram_act_wdata1	32	Output	Write data of the port 1 in the activation SRAM.
sram_act_rdata1	32	Input	Read data of the port 1 in the activation SRAM.

Simulation & Synthesis

1. Waveform
   There are two ways to view the waveform. One is to use Cadence NCVerilog, and the other is to use Synopsys VCS. Refer to "CT Verilog Series 02-1 - Update about Waveform Format" (https://eeclass.nthu.edu.tw/media/doc/117035) for instructions on how to utilize them.

   For this homework, our testbench uses Cadence NCVerilog, but the VCD format does not support packed-array debugging. Therefore, if you want to view the waveform using another method, you can modify the testbench by changing the code below the line "// ===== Set simulation info ===== //" to the FSDB format of Synopsys VCS, and modify the Makefile accordingly.
2. RTL behavior simulation
   Please make sure the result of the RTL behavior simulation is correct. Figure 5 shows the simulation message when the validation is passed. You should modify the file **sim_rtl.f** with proper file paths.
   - RTL simulation commands:

```
cd sim/
make sim
```

```
Reset System
Compute start
Compute finished, start validating result...
====================
FC1 input activation [PASS]
FC1 output activation [PASS]
>>> Congratulation! FC1 result are correct
  [RTL simulation]
Clock Period: 12.00 ns,Total cycle count:        1603 cycles
====================
Simulation finish
Simulation complete via $finish(1) at time 20478 NS + 3
./lenet_tb.v:203         $finish;
ncsim> exit
```

Figure 5: Simulation passes!

3. Synthesis

   While synthesis is not required for this assignment, we recommend you to give it a try. Refer to the **Spyglass tutorial** first to see if the design can be synthesized. You may modify the Verilog file names and clock period (**cycle**) in **syn/synthesis.tcl** accordingly.

   ➢ Synthesis commands:

```
cd syn/
dc_shell -f synthesis.tcl
```

   The synthesis script also produces timing and area reports. Make sure the timing slack is **MET**.

```
clock clk (rise edge)                                  10.0000    10.0000
clock network delay (ideal)                             0.0000    10.0000
sram_act_wdata0_reg[19]/CK (DFFQXL)                     0.0000    10.0000 r
library setup time                                     -0.0727     9.9273
data required time                                                 9.9273
--------------------------------------------------------------------------
data required time                                                 9.9273
data arrival time                                                 -8.0147
--------------------------------------------------------------------------
slack (MET)                                                        1.9126
```

Figure 6: Design Compiler timing report.

Grading Policy

1. RTL simulation passes (70%)
2. Report (30%)

Report

1. Design concept:
   - Explanation of the overall hardware architecture and block diagram of each component.
   - State diagram and its detailed description (if any).
   - Explanation of the dataflow.
   - You may write the report in Chinese.
2. Result:
   You should paste the screenshot of your RTL simulation result.
3. END_CYCLE (optional)
   If you need a larger END_CYCLE than the default, please let us know what value you are using.
4. Others (optional)
   - Suggestions or comments about this class to teacher or TA.

Submission

1. Please submit the following files to EECLASS (20-point penalty if not following the rules).
   - lenet.v # Includes top module **lenet** and all other modules. Please integrate all your code into this single file, excluding SRAM_weight_16384x32b.v and SRAM_activation_1024x32b.v.
   - hw2_report_<*student_id*>.pdf # E.g., `hw2_report_0123456789.pdf`. Use the template we provided.
2. **We will compare your code with that of other students this year and previous years, as well as online resources, to detect plagiarism. Please do not engage in any form of plagiarism or academic misconduct.**
3. **DO NOT** compress submitted files!
4. **Overdue is strictly forbidden!**

Appendix

Pipeline architecture:

We **will not** grade performance in HW2, but we will consider it in HW3. The following instructions are for those who want to achieve a better performance. Pipeline is a common technique to improve the throughput with a higher clock rate. For example, in Figure 8, a combinational circuit with flip-flops can be partitioned into two or more pipeline stages. The clock rate can increase with a more significant hardware area (i.e., additional flip-flops between stages).

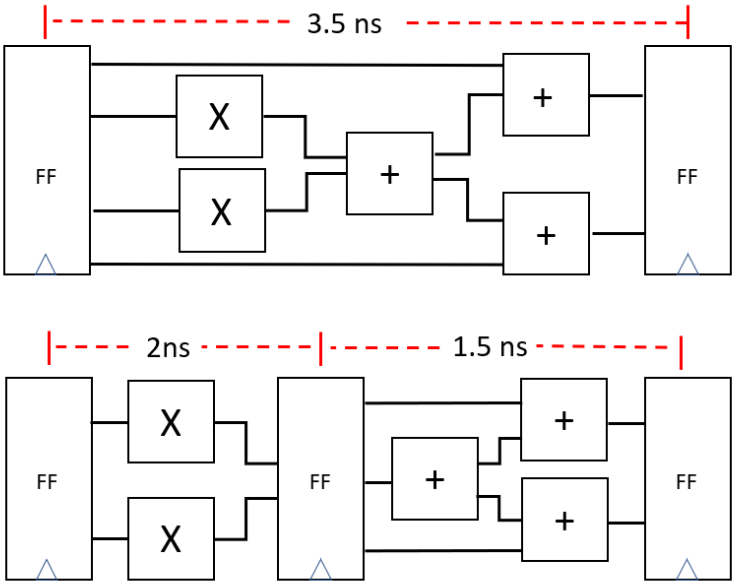

Figure 8: Simple pipeline concept.

Data arrangement in SRAMs:

You don't need to prepare any pattern files in this homework, since we prepared all three pattern files (**weights.csv**, **fc1_golden.csv**, **fc1_input.csv**) for you. However, you should trace the testbench code to know where to place them. The following instructions are for those who want to generate their own pattern files from scratch or have a better understanding of pattern files.

The first one is **weights.csv**, which consists of all quantized weights. You need to process and integrate the weight files in **parameters/weights** of HW1 (also refer to Figure 1, Table 1, and Table 2).

Note: weight.csv contains 15760 lines; each line has a 32-bits HEX data (in ASCII format for `$readmemh()`).

Table 1: Mapping between weights.csv and original quantized weight files.

Line # of weights.csv	**Original filename**
0 – 59	conv1.conv.weight.csv
60 – 1019	conv3.conv.weight.csv
1020 – 13019	conv5.conv.weight.csv
13020 – 15539	fc6.fc.weight.csv
15540 – 15749	output.fc.weight.csv
15750 – 15759	output.fc.bias.csv

Table 2: The comparison of weight files.

Processed weight file		**Original weight files**	
Line #	Data (HEX)	Line #	Data (DEC)
Filename: weights.csv		Filename: conv1.conv.weight.csv	
0	4A32EBE9	0	-23
		1	-21
		2	50
		3	74
1	000000D2	4	-46
		BLANK	0
		BLANK	0
		BLANK	0
……			
Filename: weights.csv		Filename: conv5.conv.weight.csv	
1020	FAFA05FE	0	-2
		1	5
		2	-6
		3	-6
1021	0506FDF9	4	-7
		5	-3
		6	6
		7	5
……			
Filename: weights.csv		Filename: output.fc.bias.csv	
15750	FFFFFF0A	0	-246
15751	000000AC	1	172
15752	0000005F	2	95
15753	000000B5	3	181

The second one is **golden.csv** (or fc1_golden.csv for this homework), which consists of quantized activations. You need to process and integrate the files in **parameters/activations/img0** (also refer to Figure 2, Table 3, and Table 4).

Note: The golden file for a complete model contains 753 lines. Each line has 32-bits HEX data. Table 3 and Table 4 are illustrative examples. For this homework, all values except the required activations are set to 0.

Table 3: Mapping between golden.csv and original activation files.

Line # of golden.csv	**Original filename**
0 – 255	conv1/input.csv
256 – 591	conv3/input.csv
592 – 691	conv5/input.csv
692 – 721	fc6/input.csv
722 – 742	output/input.csv
743 – 752	output/output.csv

Table 4: The comparison of activation files.

Processed activation file		**Original activation files**	
Line #	Data (HEX)	Line #	Data (DEC)
Filename: golden.csv		Filename: fc6/input.csv	
692	00000000	0	0
		1	0
		2	0
		3	0
693	00005000	4	0
		5	80
		6	0
		7	0
……			
Filename: golden.csv		Filename: output/input.csv	
722	00000054	0	84
		1	0
		2	0
		3	0
723	00000063	4	99
		5	0
		6	0
		7	0

The last one is **input.csv** (or **fc1_input.csv** for this homework), which is like **golden.csv**, but it only contains the input of the first layer. In this homework, we provide **fc1_input.csv** which contains input of fc1 from the fc6/input.csv (also refer to Figure 2, Table 5, and Table 6).

Note: input_fc1.csv contains 722 lines. Each line has 32-bits HEX data. For this homework, all lines other than input activation of the fc1 are zeros.

Table 5: Mapping between fc1_input.csv and original activation files

Line # of fc1_input.csv	**Original filename**
692-721	fc6/input.csv

Table 6: The comparison of fc1 input files.

Processed activation file		**Original activation files**	
Line #	Data (HEX)	Line #	Data (DEC)
Filename: fc1_input.csv		Filename: fc6/input.csv	
692	00000000	0	0
		1	0
		2	0
		3	0
693	00005000	4	0
		5	80
		6	0
		7	0