

## Homework 4: LeNet Accelerator Engine

Submission Due Dates:

2024/5/7 23:59

### Objective

In this homework, you will complete the CNN accelerator for the quantized LeNet model from HW3.

### Model architecture

In Homework 3, we implemented an accelerator engine with only the first fully connected layer. In this homework, you will need to finish the accelerator engine to compute all five layers of LeNet in Verilog. The CNN model is listed for your reference:

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. A testbench `./sim/lenet_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 each layer's activations. You can modify and enhance its debugging capability accordingly. **But note that we will use the original testbench to justify your design.**
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 32-bit memory accesses of separate addresses in the same clock cycle**.
3. Figure 1 shows the layout of the weight SRAM. In **Conv1 and Conv2 weight layers**, we only **store five 8-bit weights in every two addresses**, which is possible to **access five weights in one clock cycle**. Since Conv3 layer can be treated as a fully-connection layer, for **Conv3, FC1, and FC2 weight layers**, we store **four 8-bit weights in each address**. For **FC2 biases**, **one 32-bit bias is stored in every address**.

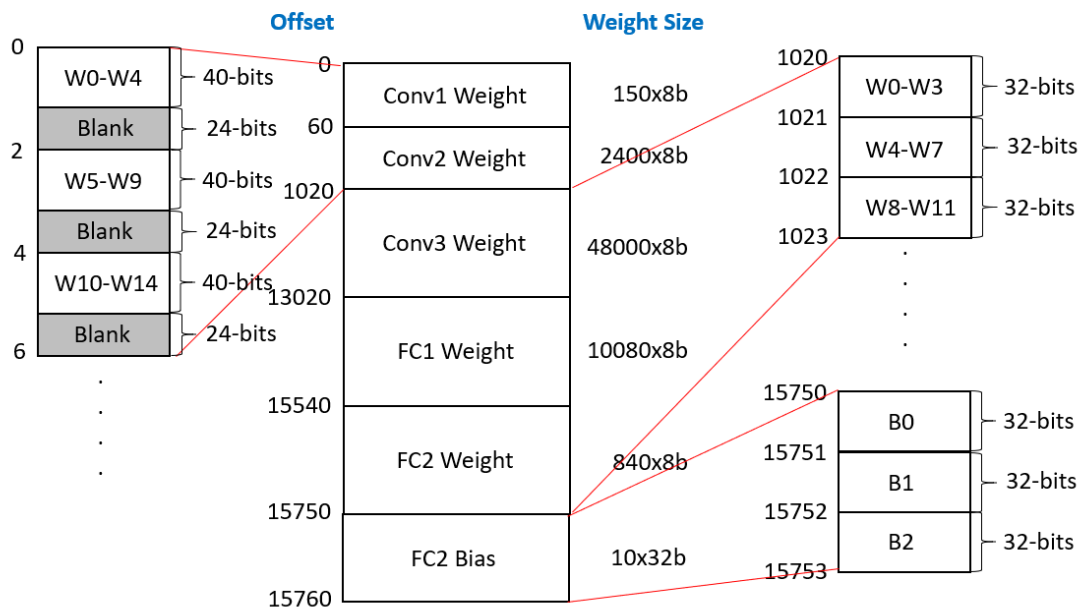


Figure 1: Layout of weight SRAM.

4. Figure 2 shows the layout of activation SRAM. The testbench will load the **input feature map from offset 0 to offset 255 in SRAM**. Each address **holds four 8-bit inputs**. In Conv1 layer, you should place fourteen 8-bit quantized activations in every four addresses, as figure 2 shows. In this case, the computation in the following Conv2 layer may be easier. For Conv2, Conv3, and FC1 layers, you should place four 8-bit quantized data in every address for the fully-connected computation. **In the last layer, FC2, we omit the rescaling and clipping** since the classification result will not

be affected. You should keep the **32-bits activation without rescaling and clipping**.

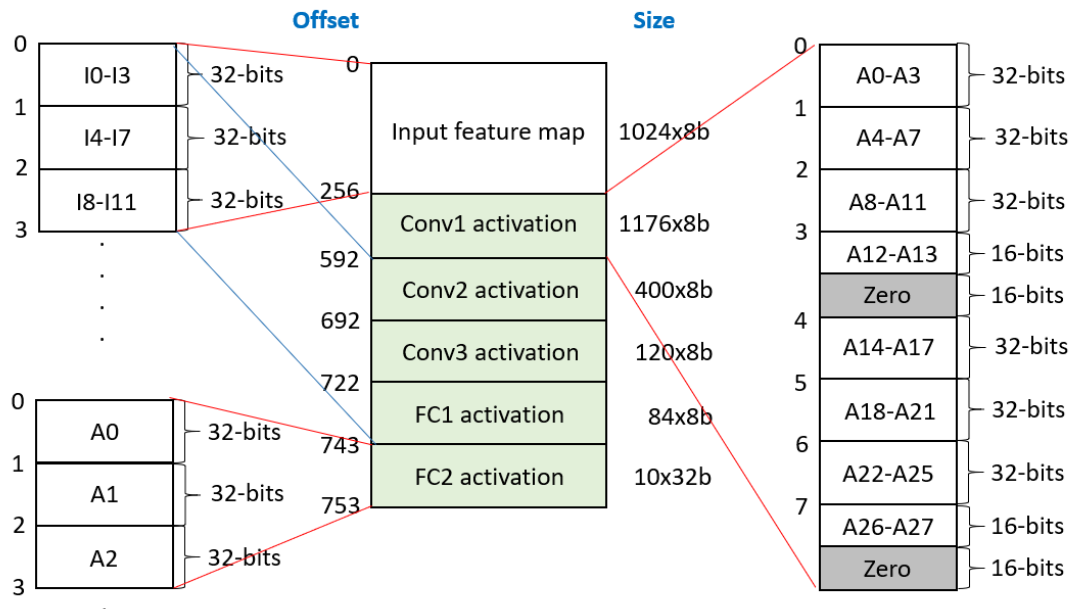


Figure 2: Layout of the activation SRAM.

5. You should use the **parameter.zip** (with image, weight, bias, quantization scale, and golden data) generated and submitted in Homework 1 to validate your design, which should be pre-processed to match the format in Figures 1 and 2 (i.e., you have to write your own tool to convert the format). It is recommended to understand the testbench before writing your own pre-processing code. (You may use either C++ or Python, or any programming language.) In addition, please check to-dos in testbench to replace your own scale and file name.

Note: 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 in the synthesis script. Make sure to add flip-flops after input data port (except for the clk port) and before output data ports.

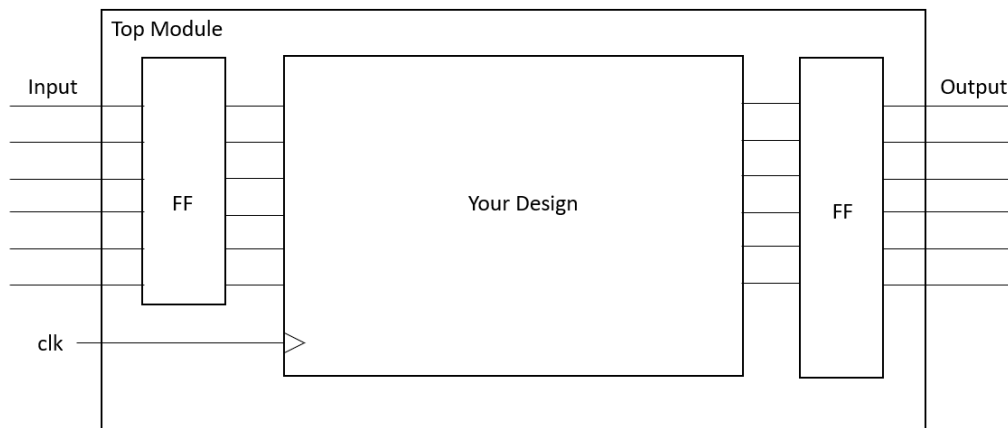


Figure 3: I/O-registered design style.

3. Figure 4 illustrates possible data reuse in convolution operations and max-pooling in CONV1 and CONV2. Typically, four 5x5 convolution operations generate 2x2 activations. The following max pool operation produces one result (see Figure 4). You may consider loading 6x6 ifmaps to improve the data reusability. Note that the dual-port SRAM allows you to load two 32-bit words (i.e., eight 8-bit ifmaps) at the same time. So it is possible to process Columns 2 to 7 while dealing with Columns 0 to 5 for further improvement.

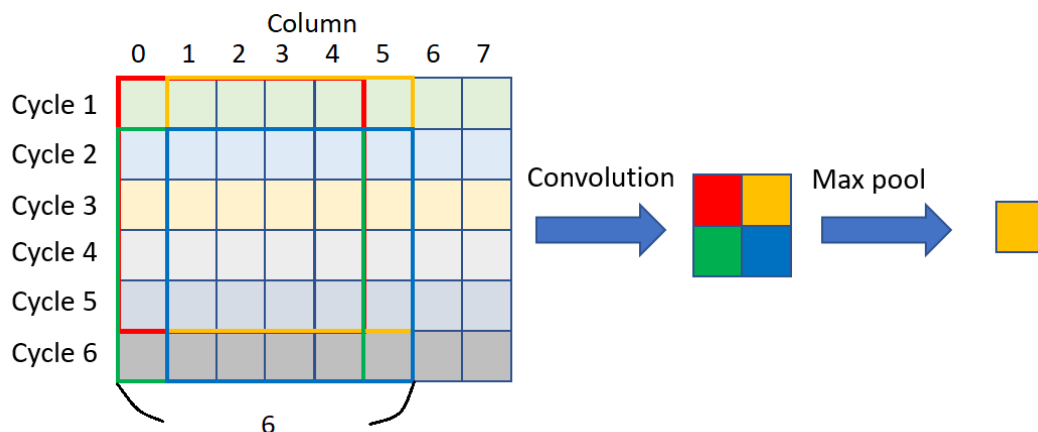


Figure 4: Data reuse for convolution operations.

#### 4. I/O port descriptions

Signals	Width	I/O	Description
clk	1	Input	Clock.
rst_n	1	Input	<b>Active-low</b> reset.
compute_start	1	Input	<b>Single-cycle active-high pulse</b> . The testbench uses this signal to inform the engine to start computing.
compute_finish	1	Output	You should <b>pull up a single-cycle active-high pulse</b> 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 port 0 in the weight SRAM.
sram_weight_wdata0	32	Output	Write data of port 0 in the weight SRAM.
sram_weight_rdata0	32	Input	Read data of port 0 in the weight SRAM.
sram_weight_wea1	4	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	32	Output	Write data of port 1 in the weight SRAM.
sram_weight_rdata1	32	Input	Read data of port 1 in the weight SRAM.
sram_act_wea0	4	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	32	Output	Write data of port 0 in the activation SRAM.
sram_act_rdata0	32	Input	Read data of port 0 in the activation SRAM.
sram_act_wea1	4	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	32	Output	Write data of port 1 in the activation SRAM.
sram_act_rdata1	32	Input	Read data of port 1 in the activation SRAM.

## Simulation & Synthesis

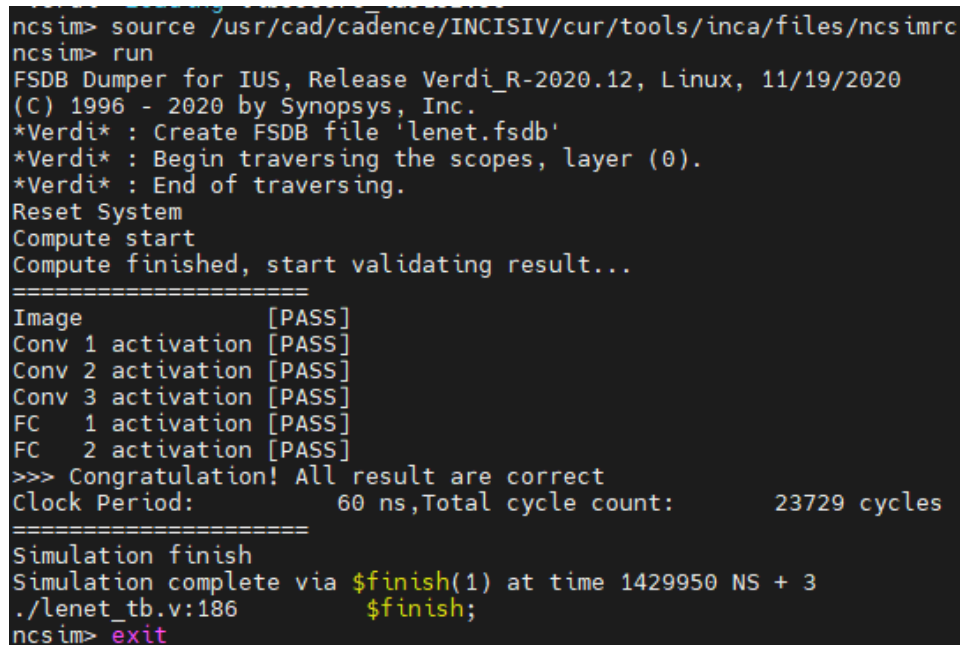
### 1. RTL behavior simulation

Before synthesis, 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 command:

```
cd sim/
make sim
```



```
ncsim> source /usr/cad/cadence/INCISIV/cur/tools/inca/files/ncsimrc
ncsim> run
FSDB Dumper for IUS, Release Verdi_R-2020.12, Linux, 11/19/2020
(C) 1996 - 2020 by Synopsys, Inc.
*Verdi* : Create FSDB file 'lenet.fsdb'
*Verdi* : Begin traversing the scopes, layer (0).
*Verdi* : End of traversing.
Reset System
Compute start
Compute finished, start validating result...
=====
Image [PASS]
Conv 1 activation [PASS]
Conv 2 activation [PASS]
Conv 3 activation [PASS]
FC 1 activation [PASS]
FC 2 activation [PASS]
>>> Congratulation! All result are correct
Clock Period: 60 ns, Total cycle count: 23729 cycles
=====
Simulation finish
Simulation complete via $finish(1) at time 1429950 NS + 3
./lenet_tb.v:186 $finish;
ncsim> exit
```

Figure 5: Simulation pass message

### 2. Synthesis

Please refer to the **Spyglass tutorial** to ensure that your design is synthesizable. You may modify the Verilog file names and clock period (**cycle**) in **syn/synthesis.tcl** accordingly.

➤ Synthesis command:

```
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 screenshot

### 3. Gate-level simulation

Before doing gate-level simulation, please modify the clock period (**CYCLE**) to meet the synthesis clock period in **sim/lenet\_tb.v**. You may add 1 to 3 ns to the synthesis clock period when encountering setup time violation. For example, if you set the synthesis clock constraint to 10 ns, you may set the clock period in the testbench to 12 ns to prevent setup time violation in gate-level simulation.

Gate-level simulation command:

```
cd sim/
make syn
```

## Grading Policy

1. RTL simulation pass (40%)
2. Spyglass report to show your design is synthesizable (10%)
3. Gate-level simulation pass (20%)

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

4. Report (10%)
5. Performance ranking (20%)

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

Please fill following items to this form: <https://reurl.cc/YVzMqa>

(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  $\mu m^2$  )

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

```
=====
Image [PASS]
Conv 1 activation [PASS]
Conv 2 activation [PASS]
Conv 3 activation [PASS]
FC 1 activation [PASS]
FC 2 activation [PASS]
>>> Congratulation! All result are correct
Clock Period: 12 ns Total cycle count: 23729 cycles
Simulation finish Clock period Latency
Simulation complete via $finish(1) at time 285990580 PS + 0
./lenet_tb.v:196 $finish;
ncsim> exit
```

Figure 7: Screenshot of gate-level simulation result.



## Report

1. Design concept (Brief and concise explanation of one to two pages would be enough. You may use Chinese.)
  - Overall hardware architecture (4%)
  - Block diagram (2%)
  - Finite-state machine diagram (2%)
  - Explanation of dataflow (and data reuse) for convolution and fully connected layers (2%)

2. Result (copy this table to report)

Item	Description
RTL simulation	(PASS / FAIL)
Gate-level simulation	(PASS / FAIL)
Gate-level simulation clock period	(XXX ns)
Gate-level simulation clock latency	(XXX cycles)
Total cell area	(XXX $\mu m^2$ )

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

## Submission

1. Please submit the following files to EECLASS (or 20-point penalty if not following the rules).
  - lenet.v # Include 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.
  - lenet\_syn.v # Gate-level netlist
  - lenet\_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>\_hw4\_report.pdf # Use the template we provide.
2. **We will compare your code with that of other students, previous students who took the course, and code found online to detect plagiarism. Please do not engage in any form of plagiarism.**
3. **DO NOT** compress submitted files!

## Appendix

### Pipeline architecture:

You may want to optimize your design for better performance. Pipeline is a common technique to improve the throughput with a higher clock rate. For example, for a combinational circuit with flip-flops in Figure 8, you can partition it into two or more pipeline stages. The clock rate can be higher with a larger hardware area (i.e., additional flip-flops between stages).

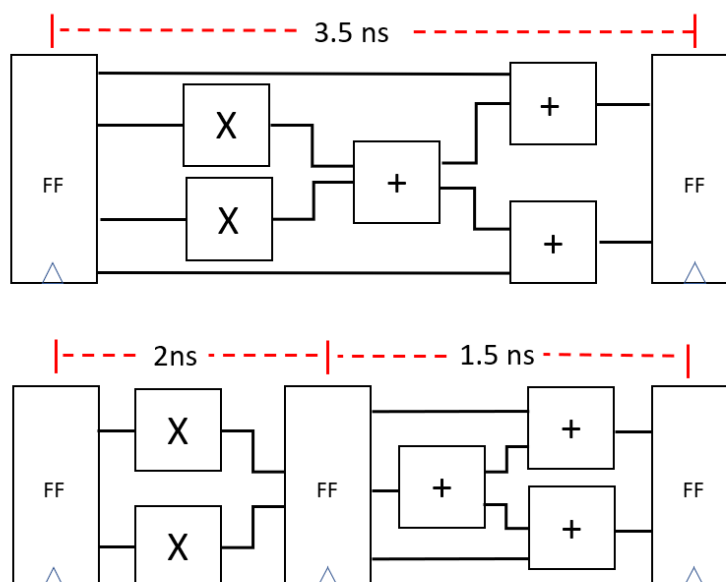


Figure 8: Simple pipeline concept.

### Data arrangement in SRAMs:

You need to prepare three pattern files for Verilog simulation (please trace the testbench code to know where to place them). First one is **weights.csv**, which consists of all quantized weights. You need to process and integrate the weight files in **parameters/weights** of HW2 (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 of 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	conv5conv.weight.csv

13020 – 15539	f6.fc.weight.csv
15540 – 15749	output.fc.weight.csv
15750 – 15759	output.fc.bias.csv

Table 2: Weight files data comparison

Processed weight file		Original weight files	
Line #	Data (HEX)	Line #	Data (DEC)
Filename: weight.csv		Filename: conv1.conv.weight.csv	
0	26F7EAC8	0	-56
		1	-22
		2	-9
		3	38
1	0000000F	4	15
		BLANK	0
		BLANK	0
		BLANK	0
.....			
Filename: weight.csv		Filename: conv5.conv.weight.csv	
1020	FB020203	0	3
		1	2
		2	2
		3	-5
1021	FFFE00FB	4	-5
		5	0
		6	-2
		7	-1
.....			
Filename: weight.csv		Filename: output.fc.bias.csv	
15750	0000003A	0	58
15751	FFFFFFAD	1	-83
15752	FFFFFF50	2	-176
15753	0000000D	3	13

The second one is **golden.csv**, 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: golden00.csv contains 753 lines, each line has 32-bits HEX data.**

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

Line # of golden00.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: Activation files data comparison

Processed activation file		Original activation files	
Line #	Data (HEX)	Line #	Data (DEC)
Filename: golden.csv		Filename: conv1/input.csv	
65	D48F8080	260	-128
		261	-128
		262	-113
		263	-44
66	DE111B29	264	41
		265	27
		266	17
		267	-34
... ..			
Filename: golden.csv		Filename: conv3/input.csv	
324	52250800	238	0
		239	8
		240	37
		241	82
325	6E6F6F6E	242	110
		243	111
		244	111
		245	110
326	14576B6D	246	109
		247	107
		248	87
		249	20
327	0000576B	250	107
		251	87
		BLANK	0
		BLANK	0

... ..			
Filename: golden.csv		Filename: output/output.csv	
743	FFFFA07C	0	-24452
744	FFFFD10A	1	-12022
745	FFFFD82A	2	-10198
746	FFFFD645	3	-10683

The last one is **image.csv**, which is similar to **golden.csv**, but it only contains the image part. You need to process **input.csv** in **parameters/activations/img0/conv1** (also refer to Figure 2, Table 5, and Table 6).

**Note: image.csv contains 256 lines, each line has 32-bits HEX data.**

Table 5: Mapping of image.csv and original activation files

Line # of image00.csv	Original filename
0 – 255	conv1/input.csv

Table 6: Image files data comparison

Processed activation file		Original activation files	
Line #	Data (HEX)	Line #	Data (DEC)
Filename: image.csv		Filename: conv1/input.csv	
65	D48F8080	260	-128
		261	-128
		262	-113
		263	-44
66	DE111B29	264	41
		265	27
		266	17
		267	-34