National Tsing Hua University
Department of Electrical Engineering
EE429200 IC Design Laboratory, Fall 2021

Homework Assignment #5 **(15%)**
Digit classification using CNN
Assigned on Nov 18, 2021
Due by **Dec 9, 2021**

Introduction

Convolutional neural networks (CNNs) have demonstrated state-of-the-art performance in many machine learning tasks such as image classification and speech recognition.

LeNet [1] is the primary work that uses CNN to classify images. Specifically, it is used for digit classification tasks such as reading postal codes. In this assignment, you are going to implement a CNN model as shown in **Fig. 1**, which is a modified version of the original LeNet model. Note that we only focus on the implementation of **UNSHUFFLE**, **CONV1**, **CONV2**, **CONV3** and **POOL** layers. The other layers (FLATTEN, FC1 and FC2) are provided in testbench by TA.

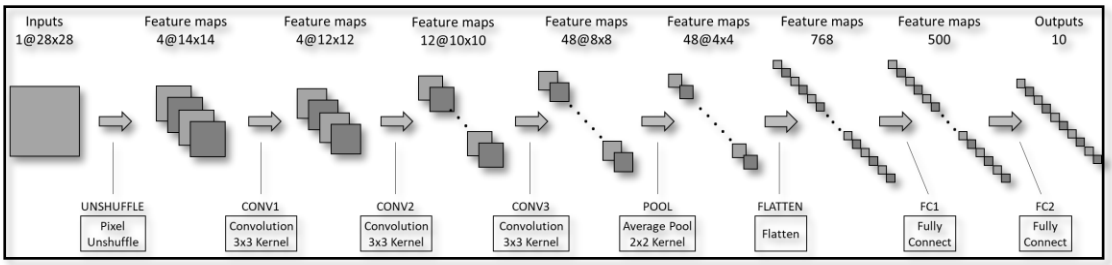

Fig. 1 *Model architecture of the modified LeNet*

Assignment Description

In this assignment, we provide two single-port SRAM groups, each group has 4 banks: **SRAM_A** (A0~A3), **SRAM_B** (B0~B3) for you to access feature maps of each layer. Another two read-only SRAMs are provided for you to access parameters (for weights and biases). The input image would be loaded from testbench in **raster scan order (z-scan order) as shown in Fig. 2**, and only **one pixel is loaded per cycle**. Then you should write the image to SRAM group A in an UNSHUFFLE way. As the data flow illustrated in **Fig. 3**, you should access SRAM_A and SRAM_B **in a rotating manner**. Note that the CONV3 layer and POOL layer should be implemented together.

More details will be mentioned in **Hardware Design** section.

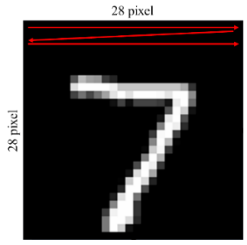

Fig. 2 *Raster scan order for loading input image*

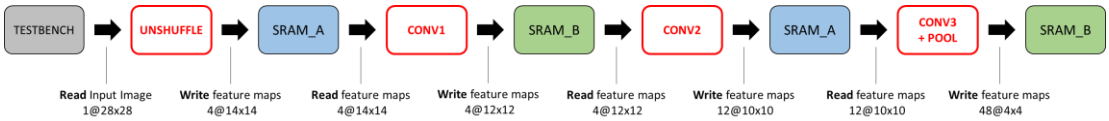

Fig. 3 *Data Flow in this assignment*

Algorithm

In this section, we will introduce the mechanism of pixel-unshuffle, convolution, average pooling and ReLU.

I. Pixel-Unshuffle

Pixel-unshuffle is a down-sample operator to reshape the input image of size $\boldsymbol{C \times H \times W}$ into 4 down-sampled sub-images of size $\boldsymbol{4C \times \frac{H}{2} \times \frac{W}{2}}$, where *C* is number of channels, *H* is image height and *W* is image width. This operator can not only improve the speed but also expand the receptive field. **Fig. 4** shows an example of how Pixel-unshuffle reshapes the input image. In this assignment, you should reshape the 1@28×28 input image into 4@14×14, and store it to SRAM group A.

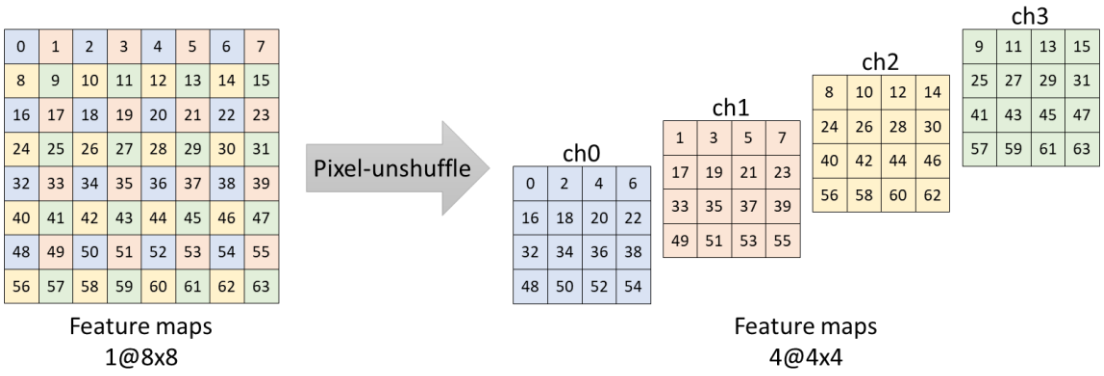

Fig. 4 *An example of unshuffling 1@8×8 input into 4@4×4 output*

II. Convolutional Layer

Convolutional layer is one of the main components in CNNs. All convolutional layers (CONV1, CONV2, CONV3) in this assignment are **3D convolution with 3×3 weight kernel**, and they are performed **without padding**. Therefore, the size of the output becomes smaller compared to the input. To understand 3D convolution clearly, let's introduce 2D convolution first.

Fig. 5 is an example of convolving 8×8 feature map with 3×3 weight kernel. We put the kernel upon the feature map, performing dot product between feature map and kernel, and sum up the 9 results to get an output pixel value. After sliding the kernel all over the image and perform the same operation as mentioned above, we can get full output result. Note that if we have *N*×*N* image and *M*×*M* kernel, the resulting size of **non-padding** convolution will be *(N-M+1)×(N-M+1)*. Therefore, the output size of this example is 6×6.

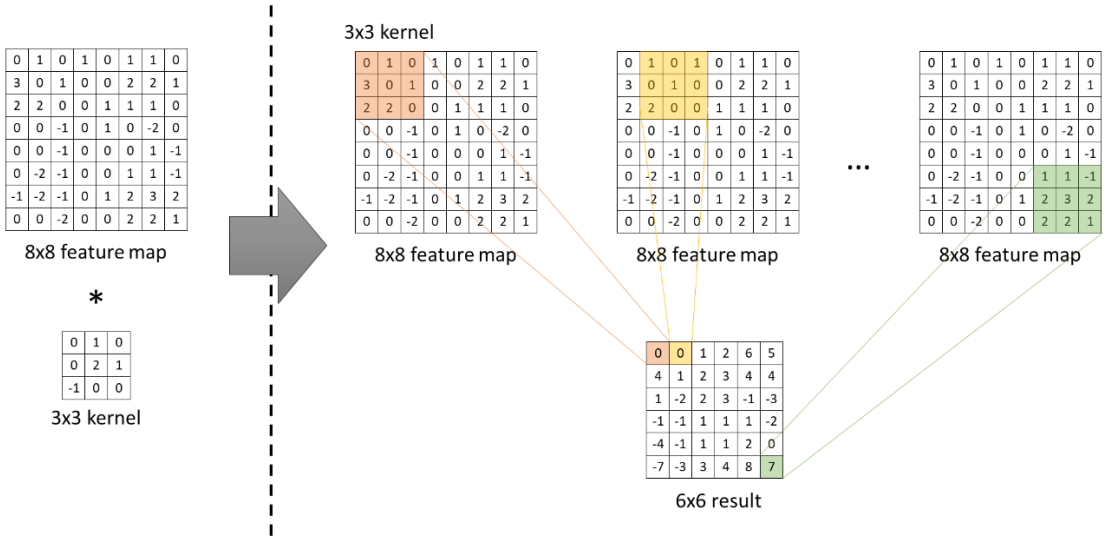

Fig. 5 *An example of convolving 8×8 feature map with 3×3 kernel.*

The difference between 2D convolution and 3D convolution is that 3D convolution has **3D input and 3D kernel**. **Fig. 6** shows computation overview of CONV1 layer. To get an output pixel value in 3D convolution, we need to **perform 2D convolution on each channel, and sum them up along the channel dimension.** That's why input feature map and weight kernel have same number of channels.

In CONV1, you need to perform 3D convolution on input feature maps 4 times, and each time using different 3D kernels, which means you need 4 sets of 3D kernels (No.0 ~ No.3) to generate 4 output feature maps. To summarize, the channel number of input feature maps is equal to channel number of 3D kernels, and the channel number of

output feature maps is equal to the number of 3D kernels.

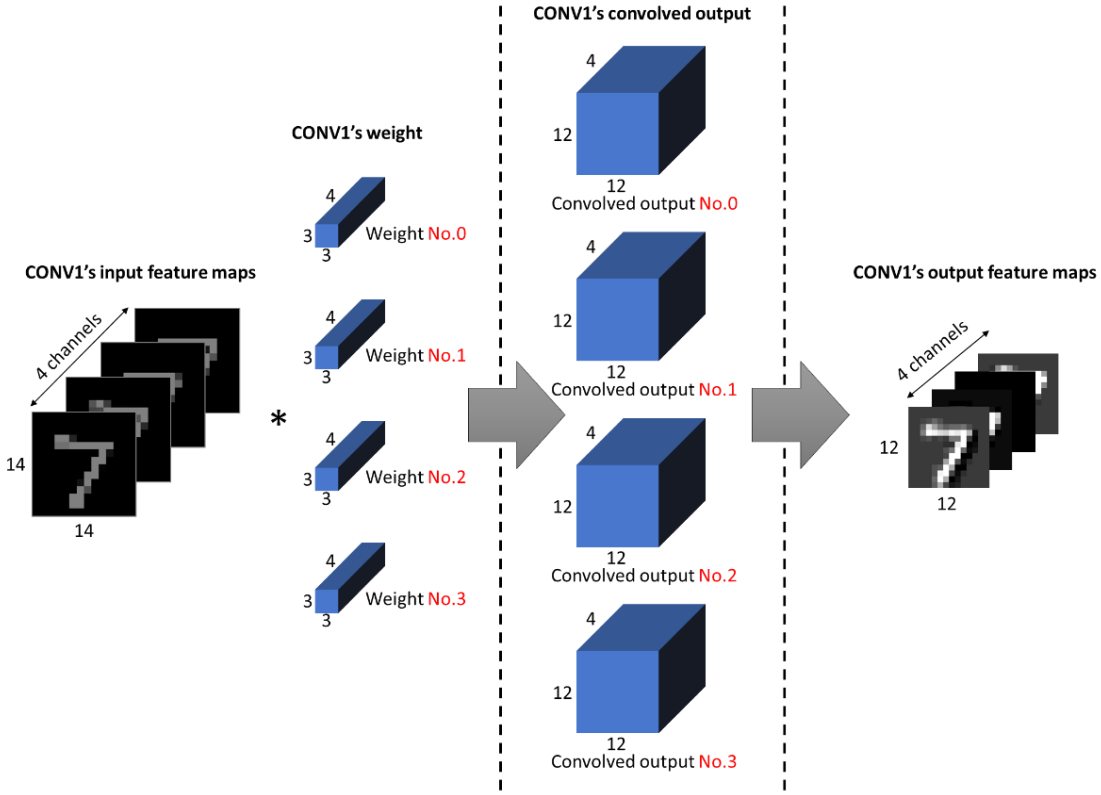

***Fig.6** Overview of CONV1's convolution operation*

For detailed operation, let's take No.0 3D convolution as an example. As illustrated in **Fig. 7**, the first step is performing 2D convolution on the input feature map with weight No.0. Then sum up the convolved output along the channel dimension. Lastly, **add bias No.0 to every pixel to get the output feature map**. We skip the details of CONV2 and CONV3 because the operations are same as CONV1.

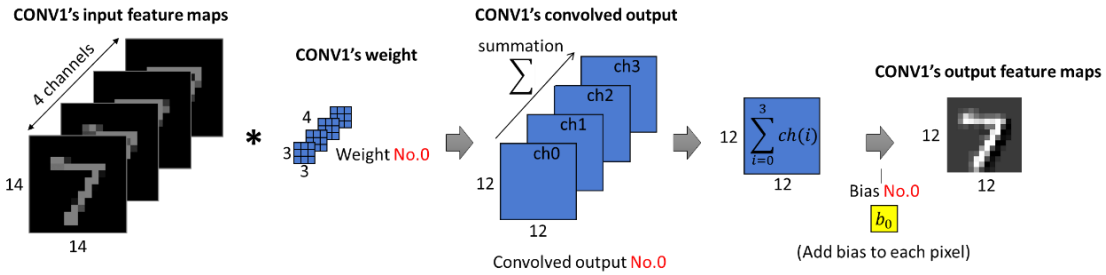

***Fig. 7** Details for No.0 3D convolution of CONV1*

III. Pooling Layer

Pooling layer is an operation of down-sampling. In this assignment, we use 2×2 average pooling that averages every 2×2 pixels, as shown in **Fig. 8**. Note that the

pooling operation is 2D, that is, we perform pooling on each channel of the input feature maps. **Fig. 9** is an example that demonstrates the dimensionality of feature maps before/after pooling operation.

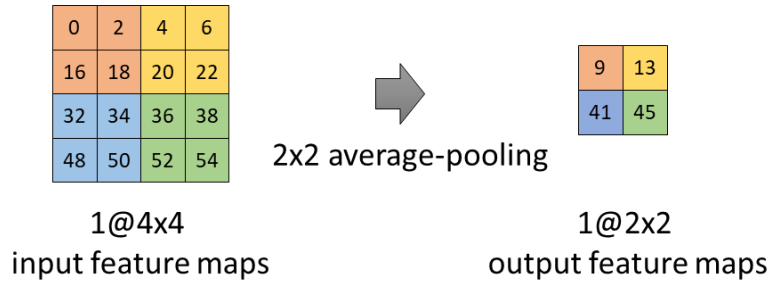

Fig. 8 *Operation of average pooling*

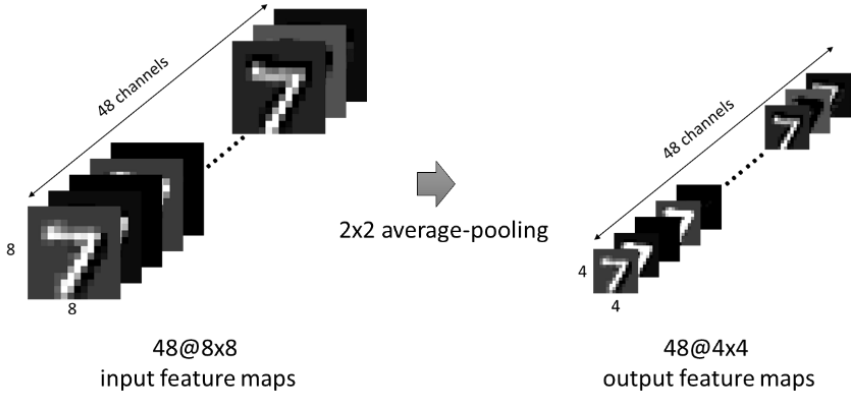

Fig. 9 *Feature map dimensionality before/after pooling*

IV. ReLU

ReLU is an activation function, and it is often used **after** convolutional layers. Its mathematical expression is $\boxed{\mathbf{f(x) = x^{+} = max(0, x)}}$, which introduces non-linearity to the CNN model. **Fig. 10** shows the plot of ReLU function. Note that in this assignment, there are **ReLUs after CONV1 and CONV2 and CONV3.**

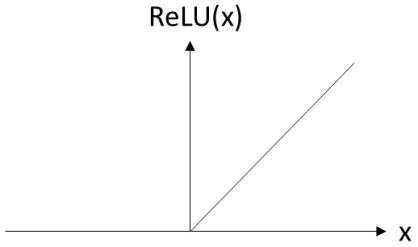

Fig. 10 *ReLU function*

Hardware Design

As mentioned in **Assignment Description** section, we want to implement CONV3 and POOL together. A naïve implementation of CONV3 and POOL is that we calculate these two layers separately, *i.e.*, calculate CONV3 and write the results to SRAM, then read them back for POOL. By merging the calculation of convolution and pooling, the data movement becomes more efficient, which is a common design technique in CNN accelerator. An illustration of merging the calculation of convolution and pooling is shown in **Fig. 11**. To get one-pixel output of pooling, it requires 16 pixels from the input feature map. Therefore, we offer a way to **access 4×4 pixels in one cycle.** Details will be addressed in the following sub-section.

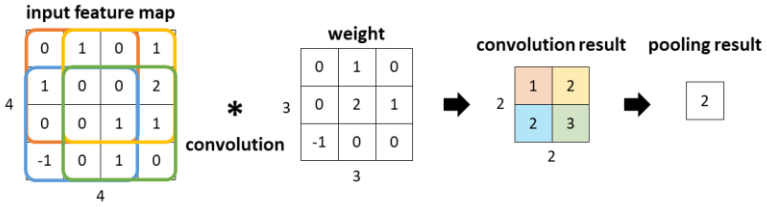

Fig. 11 *Merge convolution and pooling*

I. Details of feature map SRAM

SRAM group A and group B both have 4 banks (A0~A3 and B0~B3). As shown in **Fig. 12**, each SRAM group has size of **4@24×24 pixels**, and it is constructed by **interleaving** 4 banks. The reason why we interleave banks is that we can **access 4 banks parallelly by sending 4 addresses to 4 banks respectively**. For one address in each bank, it stores 4@2×2 pixels. Therefore, by accessing 4 banks parallelly, we can read/write 4@4×4 pixels in one cycle. In this assignment, each feature map pixel is represented in **12 bits**, and thus one address in each bank stores 16×12 = 192 bits. The data ordering of storing 4@2×2 pixels in one address is indicated in **Fig. 12**. Furthermore, each bank has 36 addresses, as also denoted in **Fig. 12**.

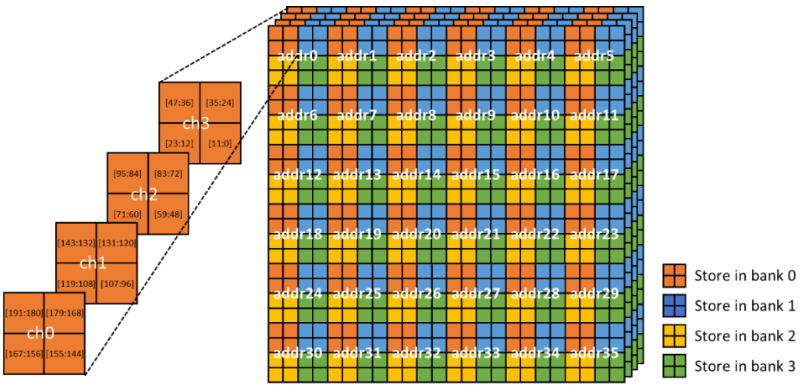

Fig. 12 *An illustration of 4-bank SRAM group*

Having the concept of SRAM group, now, we address the details of how we store output feature maps of each layer to the SRAM group by **mapping the feature maps to specific addresses**. The black box in **Fig. 13** indicates the locations of mapping each layer to the SRAM group. For simplicity, we only plot 2D view of the SRAM group, and the channel mapping is denoted by text. Since the SRAM group can only store 4 channels in one address, layers with larger number of channels, *i.e.*, CONV2 and POOL, need to partition their feature maps along channel dimensions and store them into different addresses. Remember we described in **Assignment Description** section, these layers are storing to different SRAM group (group A or group B) in a rotating manner.

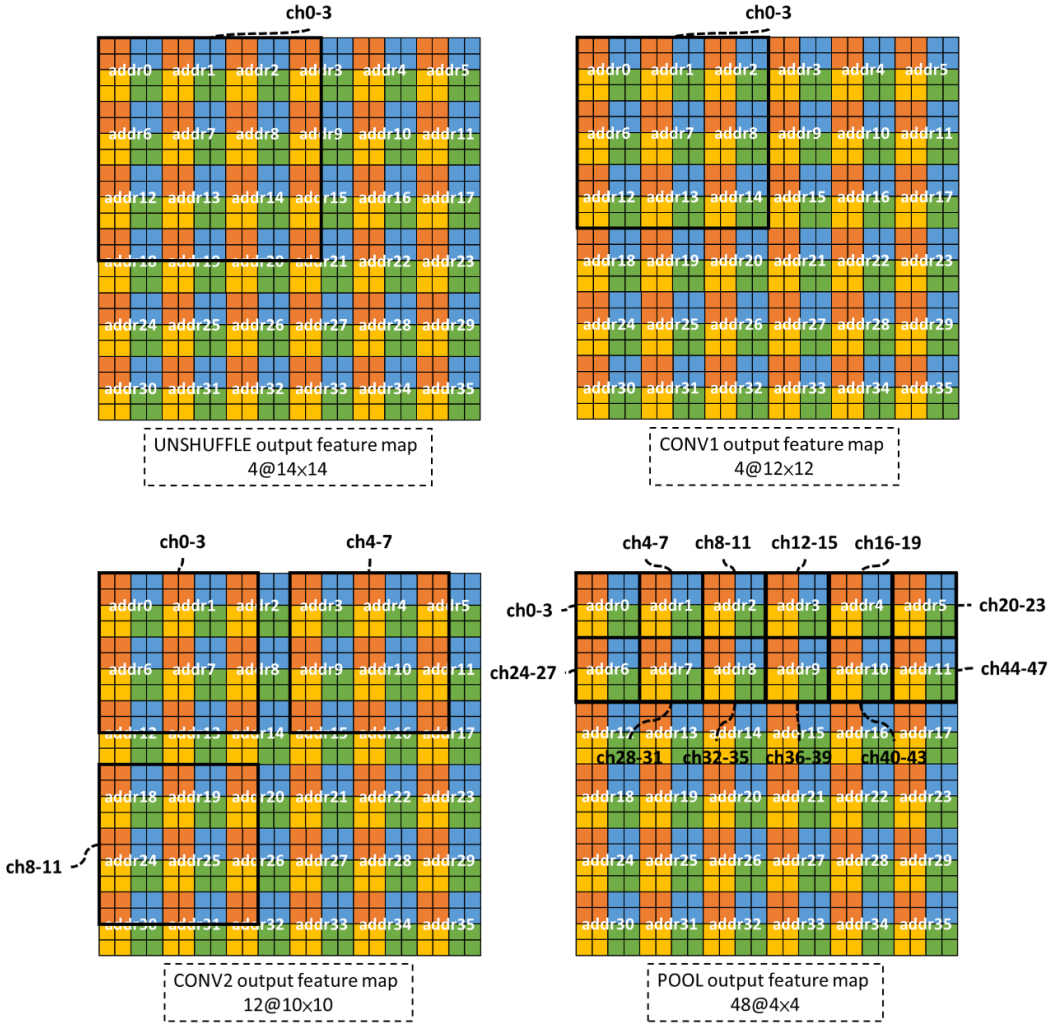

Fig. 13 *4-bank SRAM mapping for each layer*

When writing data into SRAM, you may just want to write one of the 16 pixels in a single address. In this assignment, you need to use **16-bit word mask** (each bit corresponds to 12-bit pixel data) for each SRAM bank. **Fig. 14** is an example of using 16-bit word mask. As illustrated, if you want to overwrite the data already stored in

SRAM, you should **set the corresponding word-mask bit to zero** (be careful that word-mask bit is active low), the other data will remain the same if the word-mask bit is set to one.

Data in SRAM	5	34	-123	682	1145	4	-2	-87	345	-978	345	22	14	87	-22	65
Write data	675	132	87	-314	-213	313	67	32	87	321	-22	21	-69	25	0	268
Word-mask	1	1	1	1	1	0	0	0	1	1	1	0	1	1	0	1
Data in SRAM	5	34	-123	682	1145	313	67	32	345	-978	345	21	14	87	0	65

Fig. 14 *SRAM behavior with 16-bit word mask.*

II. Details of parameter SRAM

We use **8 bits** to represent the value of model parameters. Another two SRAM are used to store them, one is for weights, and the other is for biases. For weights, we store 9 weights into one address. Specifically, these 9 weights are from a 3×3 kernel that we reshape it into 9x1 in z-scan order, as demonstrated in **Fig. 15**. For biases, we store one bias in one address.

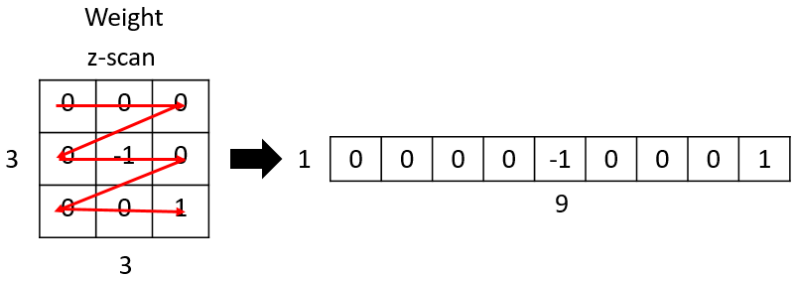

Fig. 15 *An example of reshaping a 3×3 weight kernel into 9×1.*

Table. 1 shows the location of all weights and biases needed in CONV1, CONV2, and CONV3, the ordering is based on the inference flow of the model. The detailed weight storage order of CONV1 is shown in **Table 2.** Details for CONV2 and CONV3 layers are omitted because they are similar to CONV1.

Table. 1 *Location of all weights and bias in sram_weight and sram_bias*

sram_weight	
Address	Content
0~15	Weights of CONV1 (4×4×3×3)
16~63	Weights of CONV2 (4×12×3×3)
64~639	Weights of CONV3 (12×48×3×3)

sram_bias	
Address	Content
0~3	Biases of CONV1 (4×1)
4~15	Biases of CONV2 (12×1)
16~63	Biases of CONV3 (48×1)

Table. 2 *Detailed weight storage order of CONV1*

sram_weight		
Address	Weight channel	Weight No. (output channel)
0	0	0
1	1	
2	2	
3	3	
4	0	1
5	1	
6	2	
7	3	
8	0	2
9	1	
10	2	
11	3	
12	0	3
13	1	
14	2	
15	3	

III. Details of quantization

32-bit floating point is commonly used in GPUs and CPUs, and it has a wide dynamic range which is beneficial for accuracy. In ASIC design, we typically use fixed point to represent numbers instead of floating point.

In this assignment, we use signed fixed point number (2's complement), which contains one sign bit, integer part and fractional part. **Fig. 16** is an example of 12-bit fixed point number with fractional length (bit-width of fractional part) equal to 8. Representing this number in decimal, we have $0 \cdot -2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7} + 0 \cdot 2^{-8} = 6.5703125$.

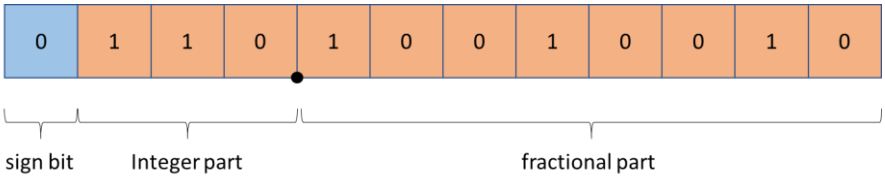

Fig. 16 *An example of 12-bit fixed point number with fl=8*

Fig.17 shows an example of fixed-point arithmetic. For multiplication C=A×B, the fractional length of C fl_C is $fl_A + fl_B$; For addition C=A+B, $fl_C = \max(fl_A, fl_B)$. Note that the decimal point should be aligned before perform addition, *i.e.*, B should be shifted left for $fl_A - fl_B = 2$ bits in this example.

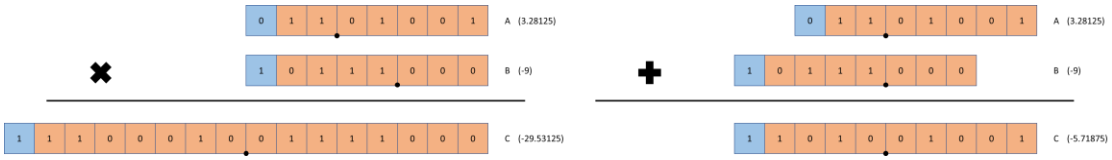

***Fig. 17** An example of fixed-point arithmetic*

In this assignment, input image and all the feature maps are represented in **12 bits with *fl* = 8**. For weights and biases, they are all represented in **8 bits with *fl* = 7**. Take fixed-point arithmetic for convolutional layer as another example. Multiply input feature maps and weights, we can get multiplied results with fractional length 8+7=15. Before adding biases, it should be left-shifted for 15-7=8 bits.

Besides, we need to round the value before quantizing the accumulated output back to 12 bits (as mentioned above, all feature maps should be quantized to 12 bits and then store into SRAM). **Fig. 18** illustrates such case. Red box indicates the 12-bits that represent the value of quantized feature maps. Note that **no matter the sign bit is 0 or 1, we just add 1 to the rounding bit**.

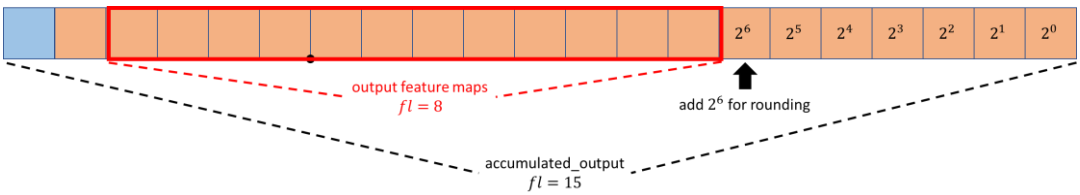

***Fig. 18** An illustration of quantizing output feature maps*

The following pseudo code may help you to understand fixed-point arithmetic and quantization of CONV layer. Be careful that **2's complement** is used in this assignment. **sign extension** may be needed to get the correct answer.

```
accumulated_output = convolved_output + (bias << 8);
q_output = (accumulated_output + 2^6) >>> 7;
if (q_output > 2047)
     q_output = 2047;
else if (q_output < -2048)
     q_output = -2048;
else
     q_output = q_output[11:0];
```

Add bias

Quantize feature map

Design Flow

The overall data flow already described in **Assignment Description** section, and illustrated in **Fig. 3**. In this section, we summarize the steps as follows.

1. Read input image from testbench
2. Write the image into SRAM group A in UNSHUFFLE way
3. Read feature maps from SRAM group A
4. Implement CONV1 layer
5. Add ReLU function at every pixel (after adding bias)
6. Quantize CONV1 feature map to 12 bits
7. Write the quantized feature maps of CONV1 to SRAM group B
8. Read feature maps from SRAM group B
9. Implement CONV2 layer
10. Add ReLU function at every pixel (after adding bias)
11. Quantize CONV2 feature map to 12 bits
12. Write the quantized feature maps of CONV2 to SRAM group A
13. Read feature maps from SRAM group A
14. Implement CONV3 layer
15. Add ReLU function at every pixel (after adding bias)
16. Perform average pooling
17. Quantize POOL feature map to 12 bits
18. Write the quantized feature maps of POOL to SRAM group B

In this assignment, we partition the whole design into three parts for you to implement it step by step. For part1, you only need to finish implementing UNSHUFFLE layer (correspond to **step 1 ~ step 2**); For part2, you need to finish implementing UNSHUFFLE + CONV1 (correspond to **step 1 ~ step 7**); For part3, the whole design (correspond to **step 1 ~ step 18**) should be implemented. In each part, **you should pull up *valid* signal after finishing writing feature maps to corresponding SRAM group**. Run simulation, then testbench will start checking your answers in corresponding SRAM group. For part3, you still can check your answers layer by layer. Refer to Appendix for details. **Note that you should unzip *bmp.zip* before running the simulation.**

Grading

1. RTL coding and simulation
   PART1 Finish **UNSHUFFLE** (pass *./part1* testbench) : 1%
   PART2 **UNSHUFFLE+CONV1** (pass *./part2* testbench): 2%
   PART3 Finish**UNSHUFFLE+CONV1+CONV2+CONV3+POOL** (pass *./part3* testbench): 3%
2. Use Spyglass to show the synthesizability (2%).
3. (**Only for those who finish whole design *i.e.* pass *./part3* testbench**) Logic synthesis with the provided scripts. Only two script files can be modified: ***0_readfile.tcl*** for adding your self-defined RTL files, and ***synthesis.tcl*** for changing the clock timing constraint TEST_CYCLE (e.g. 3.9 for 3.9 ns). Your timing slack should not be negative. The grading is based on your performance index (PI).

$$\text{Grade point: } \begin{cases} rank\ A: 7\% & if\ \boldsymbol{PI} \le 3.5 \times 10^9 \\ rank\ B: 5\% & if\ 3.5 \times 10^9 < \boldsymbol{PI} \le 7 \times 10^9 \\ rank\ C: 3\% & if\ \boldsymbol{PI} > 7 \times 10^9 \end{cases}$$

   Definition of the performance index:
   A: Total area (*e.g.* Total area: 227862)
   T: TEST_CYCLE (your timing constraint, *e.g.* 3.5)
   C: The cycle counts you completed the simulation, which is shown on the terminal (*e.g.* Total cycle count = 3712)

   ✧ **Performance index = A×T×C** (e.g. $2.96*10^9$ for the above example)

Bonus: 3% will be given to the best work in terms of the PI, and 1% to the second best.

Deliverable

Please submit all files mentioned below, and organize the file structure as follow:

1. Synthesizable Verilog functional module ***Convnet_top.v*** and all other RTL codes for your own defined modules (if any) in */hdl* folder.
2. ***sim.f*** in */sim* folder (should include all your modules and need to be executable).
3. Spyglass report file
4. A text file ***misc.txt*** indicating your completeness and performance index.
   (*i.e.* pass only part1, pass only part1 and part2, or pass all three parts. If you pass all three parts, please summarize your performance index and your A, T and C as well.)

File Structure:

- /HW5_10XXXXXXX (your student ID)
  - part1
    - /hdl
      - **Convnet_top.v**
      - **(Other Verilog files if any)**
    - /sim
      - **sim.f**
    - **spyglass report**
  - part2
    - /hdl
      - **Convnet_top.v**
      - **(Other Verilog files if any)**
    - /sim
      - **sim.f**
    - **spyglass report**
  - part3
    - /hdl
      - **Convnet_top.v**
      - **(Other Verilog files if any)**
    - /sim
      - **sim.f**
    - /syn
      - **synthesis.tcl**
      - **0_readfile.tcl**
      - **Area report**
      - **Timing report**
      - **da.log**
    - **spyglass report**
    - **misc.txt**

P.S.

1. Don't put any bmp files in the folder!!!
2. You should follow the above file structure to pack your files.
3. Remember to compress your files as HW5_10XXXXXXX.zip! Don't use .rar or other format.
4. **1%** punishment for wrong file delivery! Please redownload and double check after you upload.
5. Summit state is according to eeclass platform, please summit early.

Appendix

▪ RTL I/O Port Name Definition (for part3)

Port Name	I/O	Bitwidth	Description
clk	I	1	Clock signal
rst_n	I	1	Synchronous reset (active low)
enable	I	1	1 : Start to input bmp image
busy	O	1	0 : keep receiving input pixels 1 : stop receiving input pixels
valid	O	1	Finish writing feature maps to corresponding SRAM group
input_data	I	12	Input image pixels (one pixel per cycle, total 28x28 pixels)
sram_raddr_a0 sram_raddr_a1 sram_raddr_a2 sram_raddr_a3 sram_raddr_b0 sram_raddr_b1 sram_raddr_b2 sram_raddr_b3	O	6	**Read Address** for SRAM groups A(a0~a3), B(b0~b3)
sram_rdata_a0 sram_rdata_a1 sram_rdata_a2 sram_rdata_a3 sram_rdata_b0 sram_rdata_b1 sram_rdata_b2 sram_rdata_b3	I	192	**Read data** from SRAM groups A(a0~a3), B(b0~b3)
sram_wen_a0 sram_wen_a1 sram_wen_a2 sram_wen_a3 sram_wen_b0 sram_wen_b1 sram_wen_b2 sram_wen_b3	O	1	**Write enable** for SRAM groups (active low) A(a0~a3), B(b0~b3) 0: write SRAM mode 1: read SRAM mode
sram_waddr_a sram_waddr_b	O	6	**Write Address** for SRAM groups A(a0~a3), B(b0~b3)
sram_wdata_a sram_wdata_b	O	192	**Write data** for SRAM groups A(a0~a3), B(b0~b3)
sram_wordmask_a sram_wordmask_b	O	16	**Word mask** for SRAM groups A(a0~a3), B(b0~b3) (active low)
sram_raddr_weight	O	10	Read address for SRAM_weight
sram_rdata_weight	I	72	Read data from SRAM_weight
sram_raddr_bias	O	6	Read address for SRAM_bias
sram_rdata_bias	I	8	Read data from SRAM_bias

■ Homework File Organization

Directory	Filename	Description
/bmp	test_0000.bmp	Mnist hand write datasets (test image)
part*/hdl	Convnet_top.v	Top module
part*/sim	feature_map/*.npy golden/*.dat param/*.dat sram_model/*.v check_feature.py sim.f test_top.v	Python file for debugging Golden results of feature maps Trained weights and biases of each layer SRAM behavior model Program for you to see all layer outputs File list for RTL simulation Testbench
part3/syn	.synopsys_dc.setup *.tcl run_dc.bat	DC environment file tcl file for DC scripts Batch file for running synthesis

■ Notes

1. For debugging, we have provided *check_feature.py* for you to check all layers' output feature maps. **Fig. 19** shows how to run this python file. First, log in EE workstation wsXX, and then type ***python check_feature.py***. After entering layer and channel numbers, it will print the corresponding results. (If it doesn't work, try ***python2 check_feature.py***.)

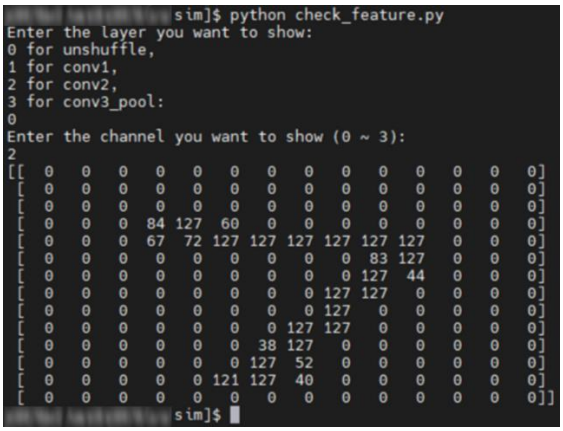

Fig.19 *An example of how to run check_feature.py.*

2. You can also implement part3 step by step. We provide a way for you to check the answers of each layer. If you finish implementing certain layer, for example, CONV2 layer (writing the results to SRAM group A), you can pull up *valid* signal. Then type the following command for simulation. You can refer the description in part3/test_top.v

```
ncverilog -f sim.f +define+TEST_CONV2
```

3. You must unzip *bmp.zip* before running the simulation.

Reference

[1] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791