

Homework Assignment #5 (15%)
Digit classification using Depthwise Separable Convolutions
Assigned on Nov 17, 2022
Due by **Dec 8, 2022**

Introduction

Convolutional neural networks (CNNs) have demonstrated state-of-the-art performance in many machine learning tasks such as image classification and speech recognition. Although conventional CNN networks are effective, they are too computing-intensive to be implemented on edge devices. Therefore, a variety of lightweight convolutional neural network models have already been proposed to achieve much better trade-offs between accuracy and computational complexity. In this homework, we will focus on one of the popular lightweight CNN model structures — Depthwise Separable Convolution (DSC).

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 lightweight 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 **CONV1_dw**, **CONV1_pw**, **CONV2_dw**, **CONV2_pw**, **CONV3** and **POOL** layers. The other layers (UNSHUFFLE, FC1 and FC2) are provided in testbench by TA.

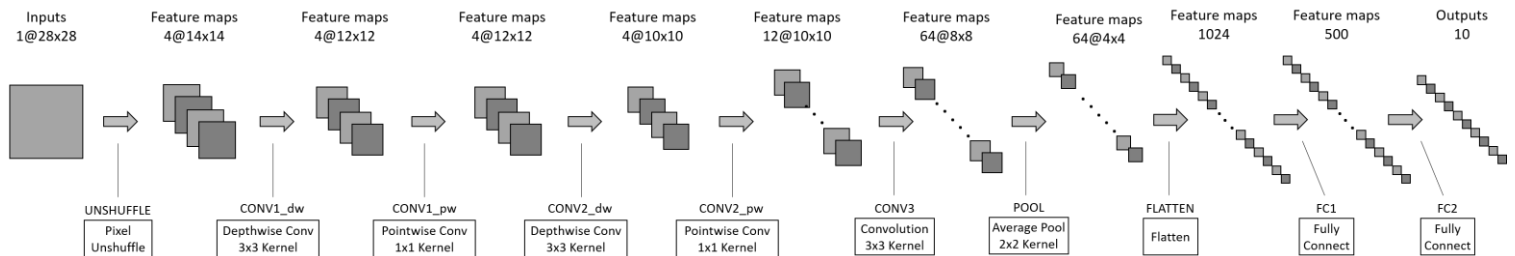


Fig. 1 Model architecture of the lightweight LeNet

Assignment Description

In this assignment, we provide two single-port SRAM groups, each group has 4 banks: **SRAM_A** (A0~A3) and **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). Since the UNSHUFFLE layer has been done by testbench, you could load the unshuffled feature maps from SRAM group A at the beginning. Then you should write the feature map to SRAM group B after the CONV1_dw operation. As the data flow illustrated in **Fig. 2**, you should access SRAM_A and SRAM_B in a **ping-pong manner**. Note that the CONV3 layer and POOL layer should be implemented together. More details will be mentioned in the **Hardware Design**

section.

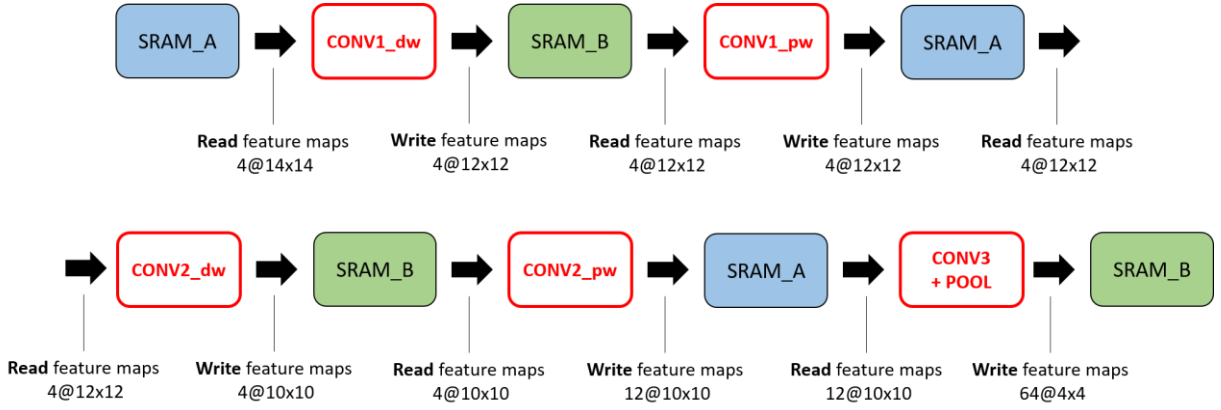


Fig. 2 Data Flow in this assignment

Algorithm

In this section, we will introduce the mechanism of **Convolution**, **Depthwise Separable Convolution (DSC)**, **Depthwise Convolution (DWC)**, **Point-wise Convolution (PWC)**, **Average Pooling** and **ReLU**.

I. Convolutional Layer

Convolutional layer is one of the main components in CNNs. Since the Depthwise Separable Convolution (DSC) is the lightweight version of it, we need to be familiar with it first.

CONV3 in this assignment are **3D convolution with 3×3 weight kernel**, and it is 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. 3 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 $N \times N$ image and $K \times K$ kernel, the resulting size of **non-padding** convolution will be $(N-K+1) \times (N-K+1)$. Therefore, the output size of this example is 6×6.

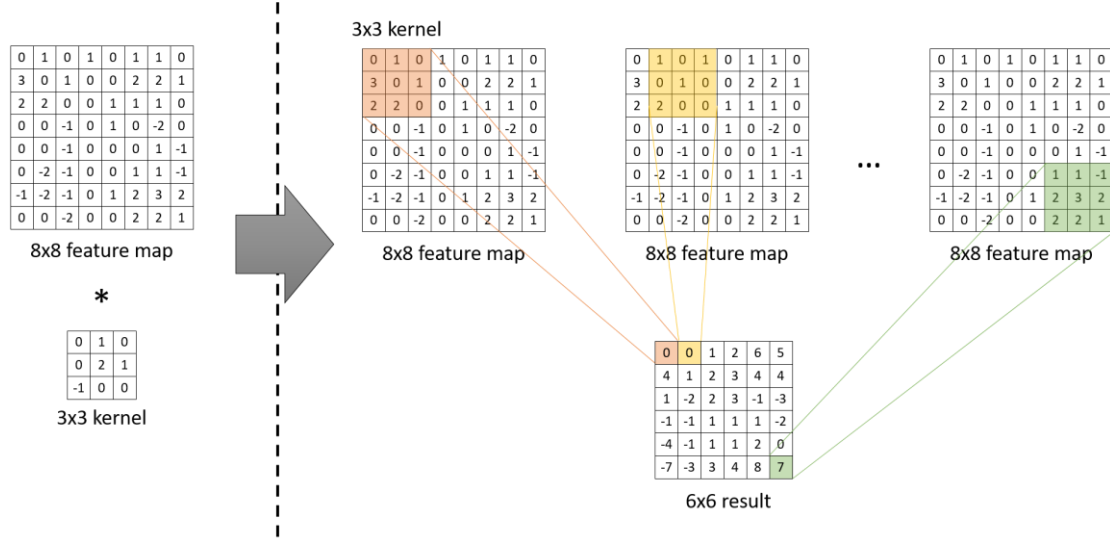


Fig. 3 An example of convolving 8×8 feature map with 3×3 kernel.

The difference between 2D convolution and 3D convolution is that 3D convolution has **3D input and 3D kernel**. **Fig. 4** shows one example of 3D convolution (input feature map: 4@12 × 12, output feature map: 12@10 × 10). 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 this convolution example, you need to perform 3D convolution on input feature maps 12 times, and each time using different 3D kernels, which means you need 12 sets of 3D kernels (No.0 ~ No.11) to generate 12 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.

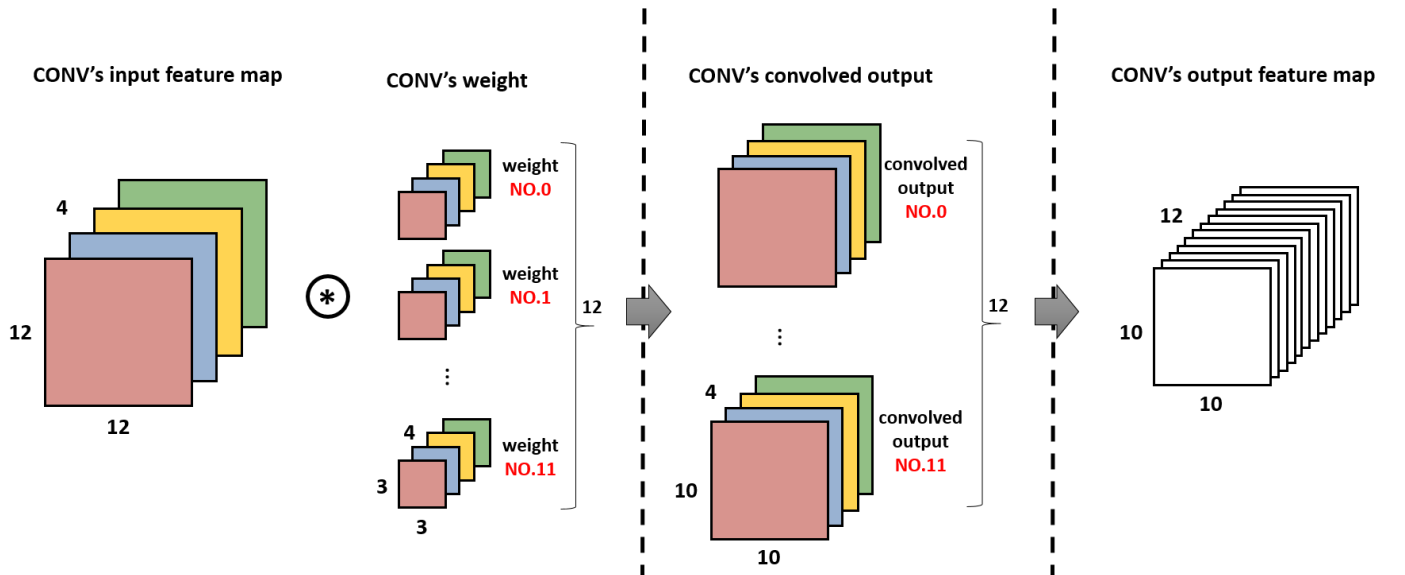


Fig.4 Overview of the convolution example

(input feature map: 4@12 × 12, output feature map: 12@10 × 10).

Comparing to standard convolution, using depthwise separable convolution considerably reduces the number of mathematical operations and the number of parameters. Let's compare standard convolution to DSC. The size of input feature map is considered as $W \times W \times N$, and the kernel size is $K \times K$ and the output feature size is considered as $W \times W \times M$; W is the spatial width and height of a square input feature map; N

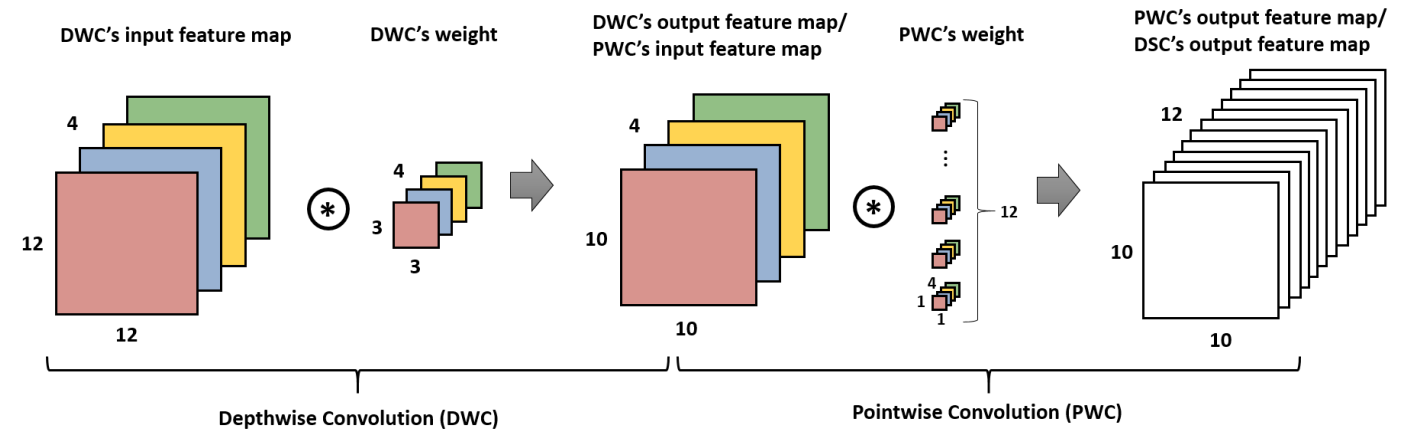


Fig. 6 Overview of the DSC example

(input feature map: $4@12 \times 12$, output feature map: $12@10 \times 10$).

is the number of input channels; M is the number of output channels.

1. number of weights:

$$W_{SC} = K \times K \times N \times M$$

$$W_{DSC} = W_{DWC} + W_{PWC} = K \times K \times N + N \times M$$

2. number of operations:

$$O_{SC} = W \times W \times K \times K \times N \times M$$

$$O_{DSC} = O_{DWC} + O_{PWC} = W \times W \times K \times K \times N + W \times W \times N \times M$$

Thus, the reduction factors on weights and operation are calculated as follows:

$$F_w = \frac{W_{DSC}}{W_{SC}} = \frac{1}{M} + \frac{1}{K^2}$$

$$F_o = \frac{O_{DSC}}{O_{SC}} = \frac{1}{M} + \frac{1}{K^2}$$

III. Depthwise Convolution (DWC) Layer

The DWC applies kernel to each input channel individually to produce the same number of output channels. Note that the DWC doesn't need to sum up the convolved results from all channels. **Fig. 7** illustrates the calculation diagrams of DWC. (input feature map: 4@12 × 12, output feature map: 4@10 × 10).

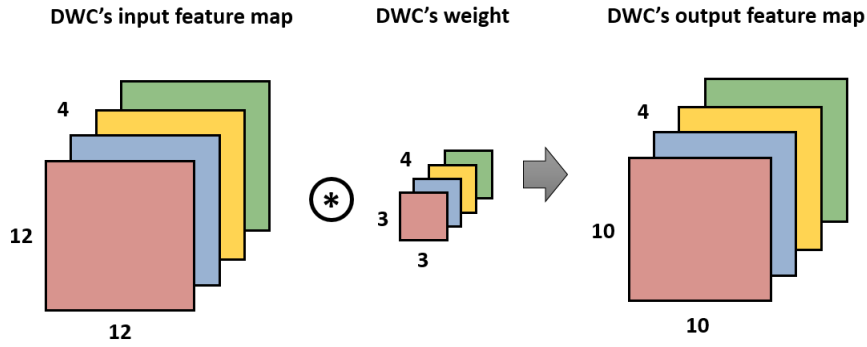


Fig. 7 Overview of the DWC example
(input feature map: 4@12 × 12, output feature map: 4@10 × 10).

IV. Pointwise Convolution (PWC) Layer

As shown in **Fig. 8** and **Fig. 9**, the PWC is actually a standard convolution with kernel size 1 × 1.

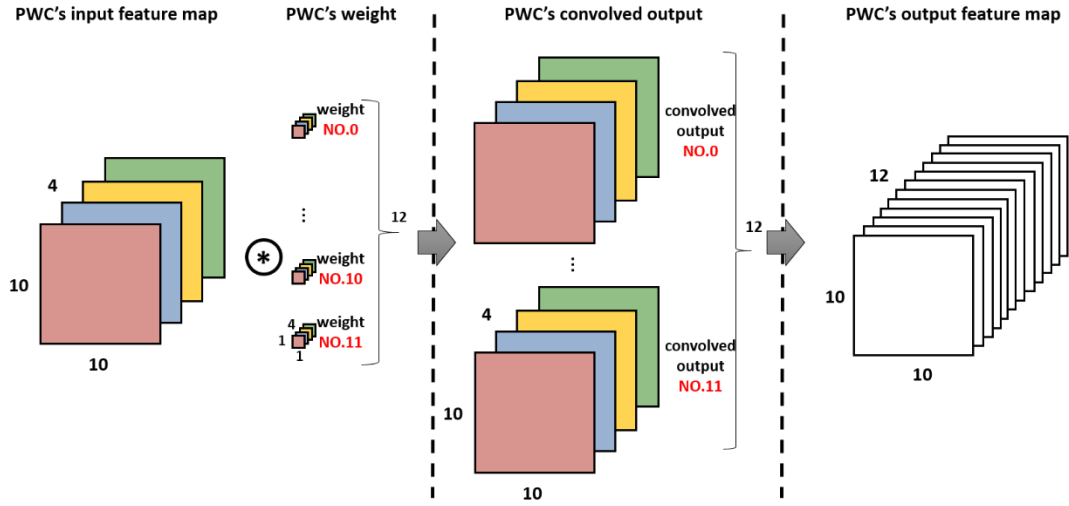


Fig. 8 Overview of the PWC example

(input feature map: 4@10 × 10, output feature map: 12@10 × 10).

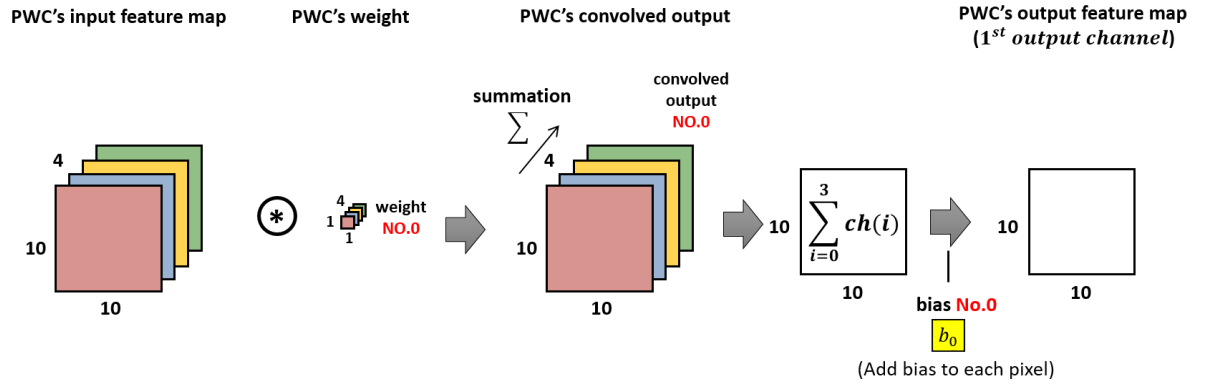


Fig. 9 Details for No.0 3D convolution of the PWC example

V. 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. 10**. Note that the pooling operation is 2D, that is, we perform pooling on each channel of the input feature maps. **Fig. 11** is an example that demonstrates the dimensionality of feature maps before/after pooling operation.

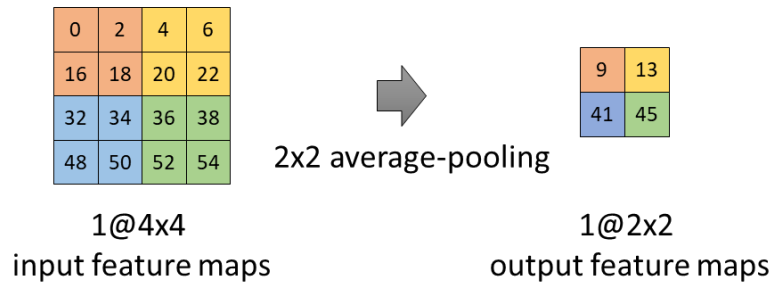


Fig. 10 Operation of average pooling

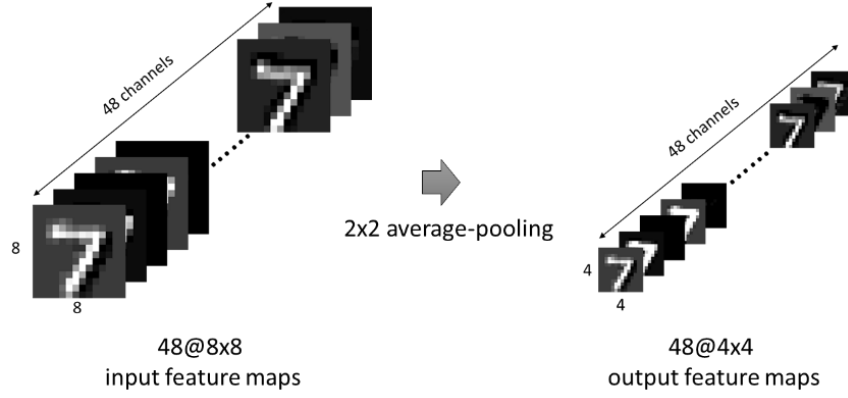


Fig. 11 Feature map dimensionality before/after pooling

VI. ReLU

ReLU is an activation function, and it is often used **after** convolutional layers. Its mathematical expression is $f(x) = x^+ = \max(0, x)$, which introduces non-linearity to the CNN model. **Fig. 12** shows the plot of ReLU function. Note that in this assignment, there are **ReLUs after CONV1_pw and CONV2_pw and CONV3**.

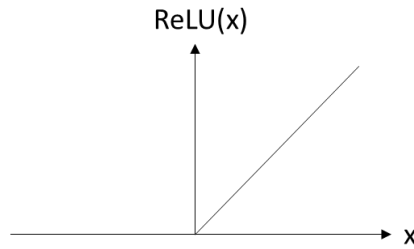


Fig. 12 ReLU function

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 the CNN accelerator. An illustration of merging the calculation of convolution and pooling is shown in **Fig. 13**. 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.

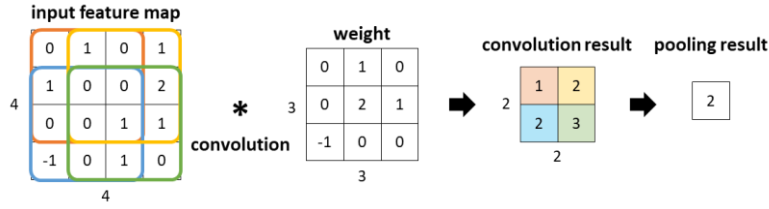


Fig. 13 Merge convolution and pooling

I. Details of feature map SRAM

SRAM group A and group B both have 4 banks (A0~A3 and B0~B3), but they have different sizes and shapes. As shown in Fig. 14, SRAM group A has size of **4@16×36 pixels**, and SRAM group B has size of **4@12×24 pixels**; both are constructed by **interleaving** 4 banks. The reason why we interleave banks is that we can **read 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 4@4×4 pixels** in one cycle. However, for the data write, all banks in the SRAM share the same write address, write data, and wordmask ports (as shown in Fig. 15), so you should at most write **4@2×2 pixels** into the specific bank by controlling the write enable signal.

In this assignment, each feature map pixel is represented in **12 bits**, and thus one address in each bank stores $16 \times 12 = 192$ bits. The data ordering of storing 4@2×2 pixels in one address is indicated in Fig. 14. Furthermore, both SRAM illustrations are also denoted in Fig. 14.

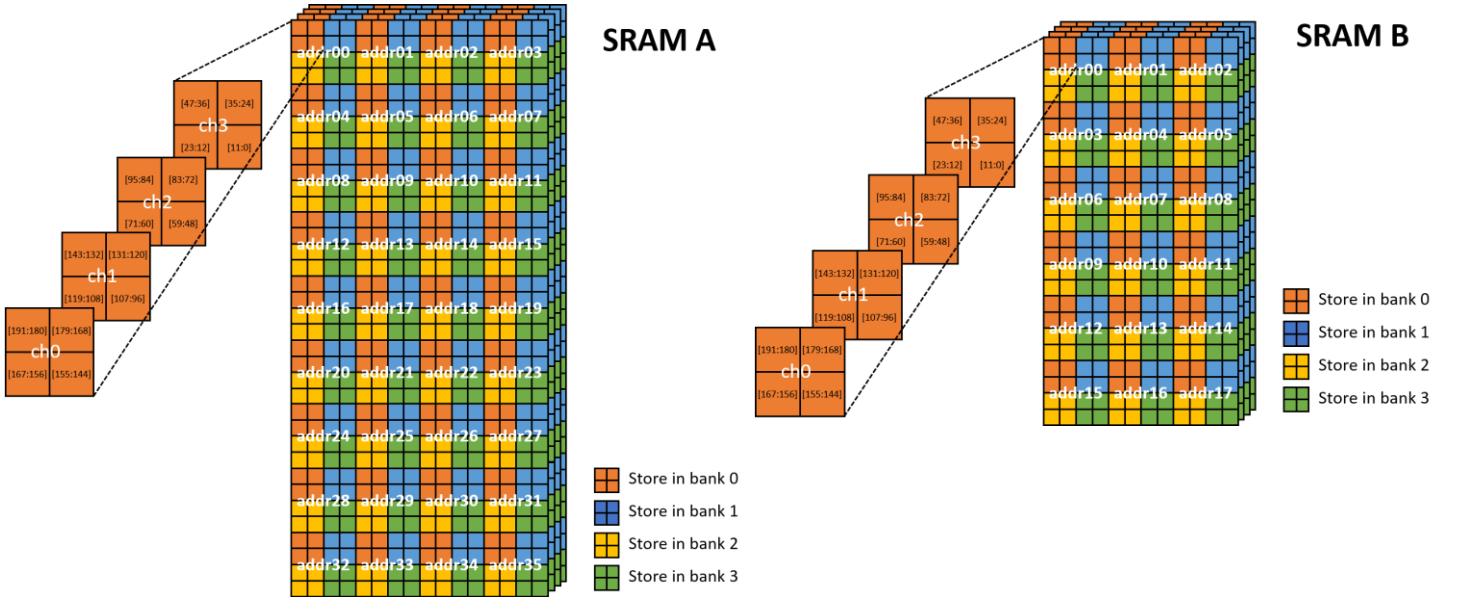


Fig. 14 An illustration of 4-bank SRAM group A and group B

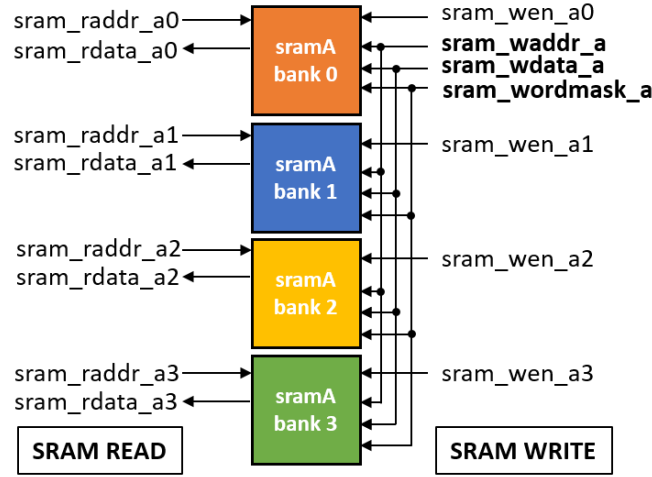
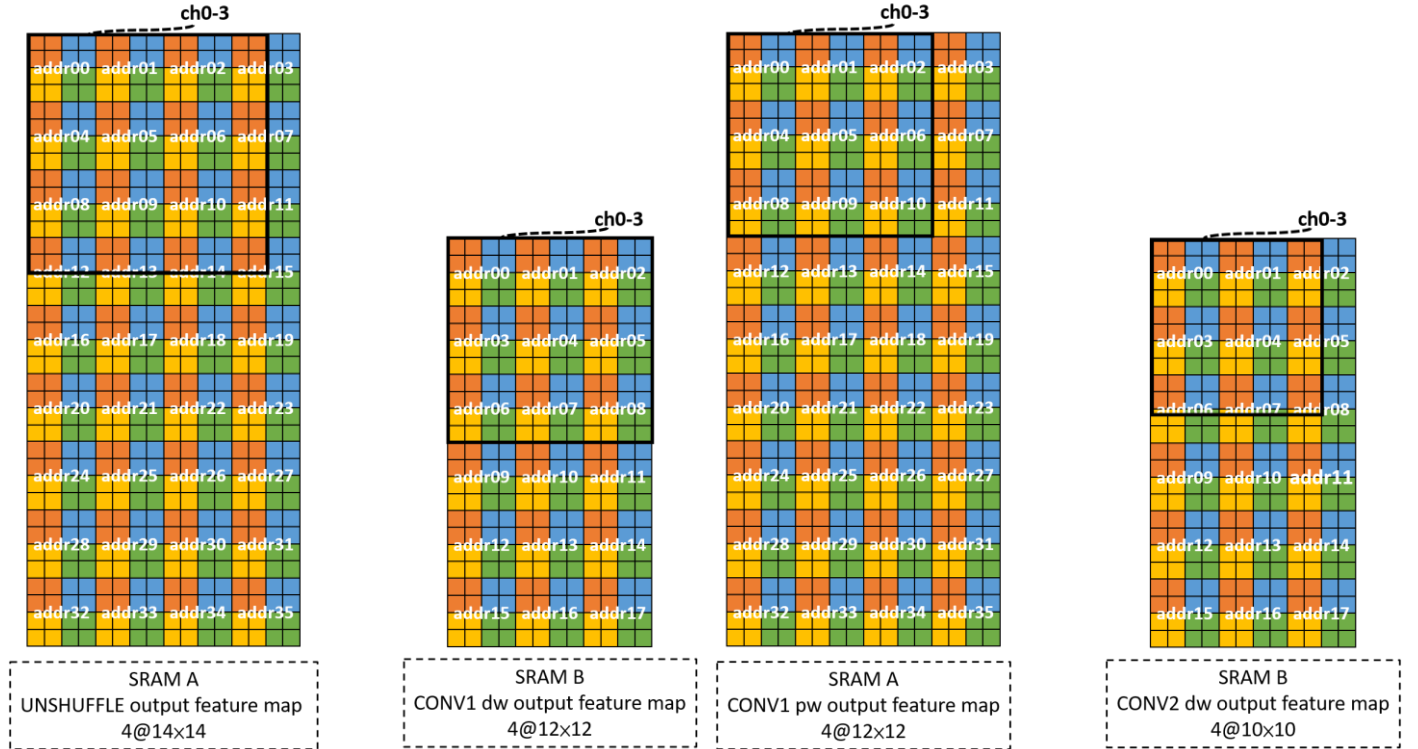


Fig. 15 Schematic of SRAM I/O ports (take sram A as example)

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. 16** indicates the SRAM mapping locations of each layer. 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_pw and CONV3_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 ping-pong manner.



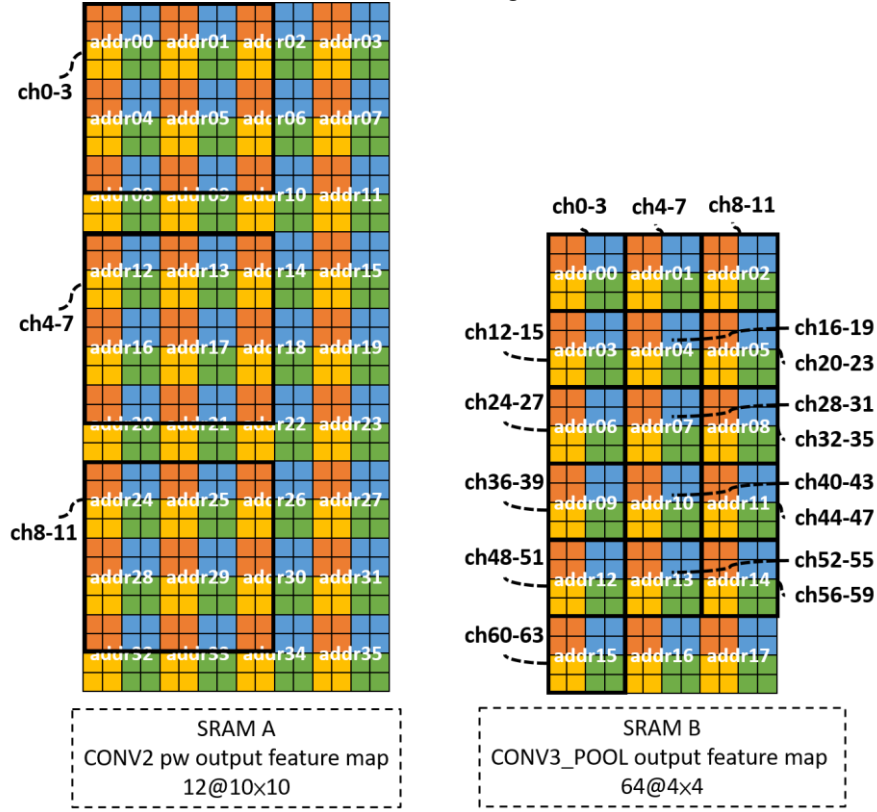


Fig. 16 4-bank SRAM A and SRAM B mapping for each layer

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**.

Fig. 17 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. 17 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 SRAMs are used to store them, one is for weights, and the other is for biases. For **DWC** and **SC weights**, we store **9 weights** into **one** address. The **Fig. 18** takes CONV1_dw as an example. Since they both use 3x3 kernels, we can easily distribute these

9 weights into 9x1 in z-scan order and store them in one address.

For **PWC** weights, because the number of weights cannot be evenly divided by 9, we store **16 weights** into **two** addresses, and the unused bytes (i.e. $18-16 = 2$ bytes) will be filled with **zeros**. As shown in **Fig. 19**, we take CONV2_pw as an example.

For all convolutions' **biases**, we store one bias in one address.

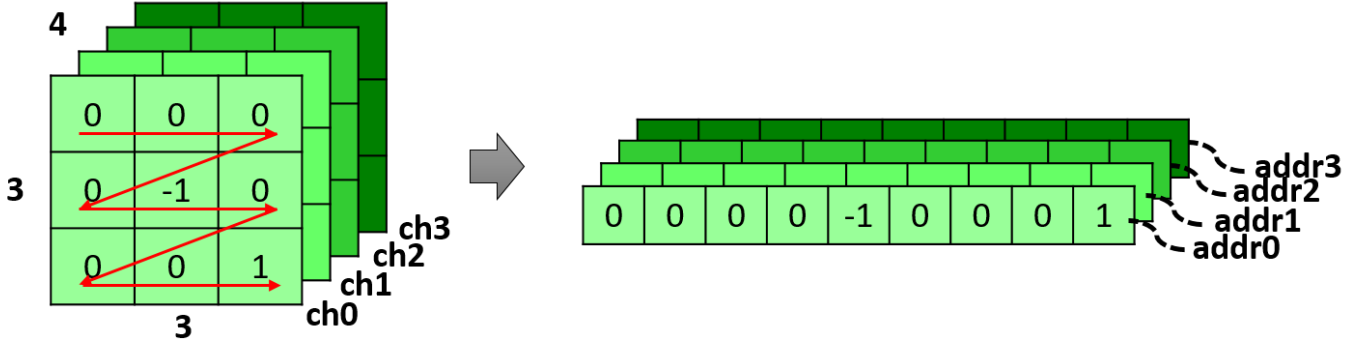


Fig. 18 An example of mapping DWC weights into SRAM (CONV1_dw)

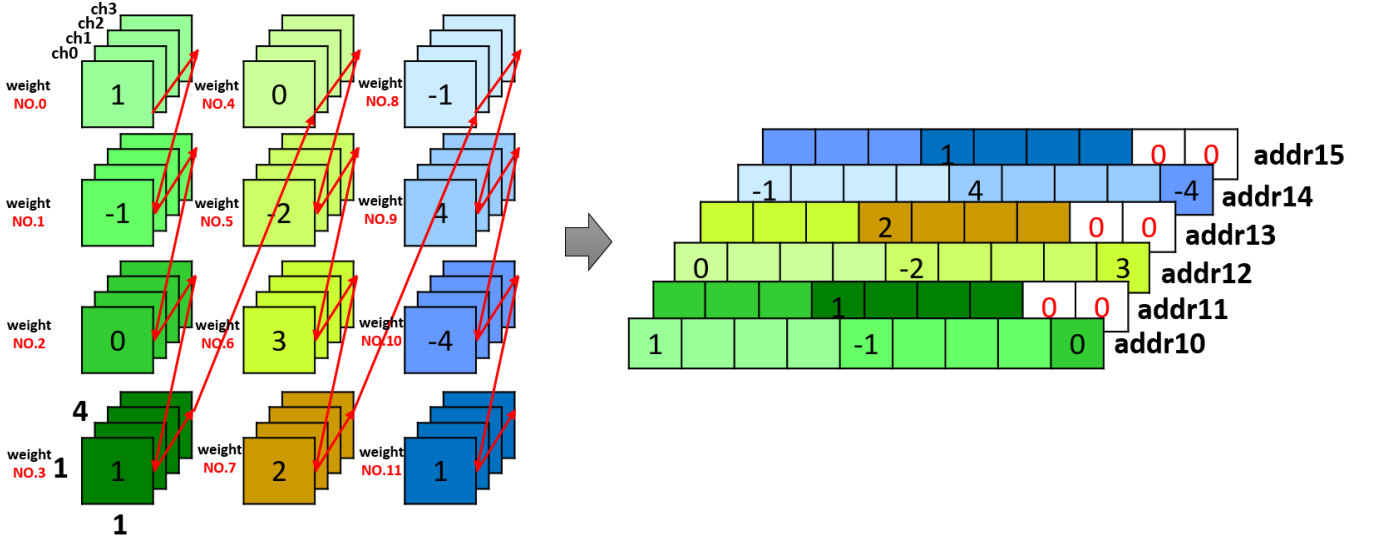


Fig. 19 An example of mapping PWC weights into SRAM (CONV2_pw)

Table. 1 shows the location of all weights and biases needed in CONV1_dw, CONV1_pw, CONV2_dw, CONV2_pw, and CONV3, the ordering is based on the inference flow of the model.

The detailed weight storage order of CONV3 is shown in **Table 2**. Details for CONV_dw and CONV_pw layers are omitted because they have been illustrated in **Fig. 18** and **Fig. 19**.

Table. 1 Location of all weights and biases in sram_weight and sram_bias

sram_weight (sram_784x72b)		sram_bias (sram_88x8b)	
Address	Content	Address	Content
0~3	Weights of CONV1_dw (4 x 3 x 3)	0~3	Biases of CONV1_dw (4 x 1)
4~5	Weights of CONV1_pw (4 x 4 x 1 x 1)	4~7	Biases of CONV1_pw (4 x 1)
6~9	Weights of CONV2_dw (4 x 3 x 3)	8~11	Biases of CONV2_dw (4 x 1)
10~15	Weights of CONV2_pw (4 x 12x 1 x 1)	12~23	Biases of CONV2_pw (12 x 1)
16~783	Weights of CONV3 (12 x 64 x 3 x 3)	24~87	Biases of CONV3 (64 x 1)

Table. 2 Detailed storage order of CONV3

sram_weight (sram_784x72b)		
address	Weight channel	Weight No. (output channel)
16~27	0~11	0
28~39	0~11	1
40~51	0~11	2
...
...
772~783	0~11	63

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. 20** 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 $0 \cdot -2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7} + 0 \cdot 2^{-8} = 6.5703125$.

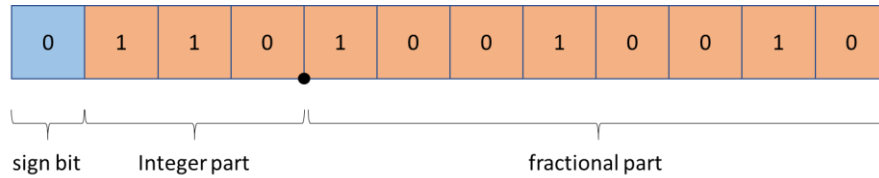
**Fig. 20** An example of 12-bit fixed point number with $fl=8$

Fig. 21 shows an example of fixed-point arithmetic. For multiplication $C=A \times B$, 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.

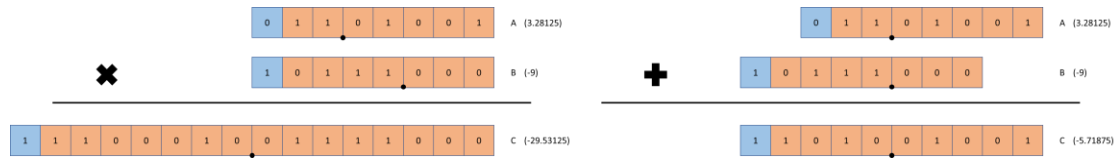


Fig. 21 An example of fixed-point arithmetic

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 back into SRAM). **Fig. 22** 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**.

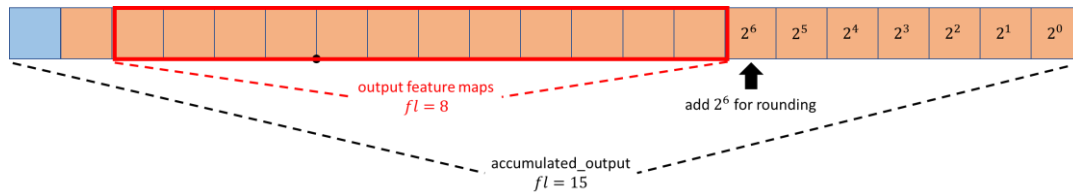


Fig. 22 An illustration of quantizing output feature maps

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); }           Add bias
q_output = (accumulated_output + 26) >>> 7;
if (q_output > 2047)
    q_output = 2047;
else if (q_output < -2048)
    q_output = -2048;
else
    q_output = q_output[11:0]; }           Quantize feature map

```

Design Flow

The overall data flow is already described in **Assignment Description** section, and illustrated in **Fig. 2**. In this section, we summarize the steps as follows.

1. Read unshuffled feature maps from SRAM group A
2. Implement CONV1_dw layer
3. Quantize CONV1_dw feature map to 12 bits
4. Write the quantized feature maps of CONV1_dw to SRAM group B
5. Read feature maps from SRAM group B
6. Implement CONV1_pw layer
7. Add ReLU function at every pixel (after adding bias)
8. Quantize CONV1_pw feature map to 12 bits
9. Write the quantized feature maps of CONV1_pw to SRAM group A
10. Read feature maps from SRAM group A
11. Implement CONV2_dw layer
12. Quantize CONV2_dw feature map to 12 bits
13. Write the quantized feature maps of CONV2_dw to SRAM group B
14. Read feature maps from SRAM group B
15. Implement CONV2_pw layer
16. Add ReLU function at every pixel (after adding bias)
17. Quantize CONV2_pw feature map to 12 bits
18. Write the quantized feature maps of CONV2_pw to SRAM group A
19. Read feature maps from SRAM group A
20. Implement CONV3 layer
21. Add ReLU function at every pixel (after adding bias)
22. Perform average pooling
23. Quantize POOL feature map to 12 bits
24. Write the quantized feature maps of POOL to SRAM group B

In this assignment, we partition the whole design into four parts for you to implement it step by step. For part1, you only need to finish implementing CONV1_dw layer (correspond to **step 1 ~ step 4**); For part2, you need to finish implementing CONV1_dw + CONV1_pw (correspond to **step 1 ~ step 9**); For part3, you need to finish implementing CONV1_dw + CONV1_pw + CONV2_dw + CONV2_pw (correspond to **step1 ~ step 18**); For part4, the whole design (correspond to **step 1 ~ step 24**) 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 part2-4, you still can check your answers layer by layer. Refer to Appendix for details.

Grading Policy

The grading policy is divide into three part: **functionality score**, **basic synthesis score**, and **PI ranking score**. The functionality score depends on if your design could pass during the pre-simulation. The basic synthesis score is for those whose design can pass **PART4 pre-simulation**, and it depends on the synthesis performance of your design. Table 2 shows detailed grading policy for each grade. The PI ranking score is for those whose **PI Grade achieve A**. Your PI ranking score will be decided by your relative rank in the class. **Fig. 23** illustrates the PI ranking score policy.

Before synthesis, you should run **Spyglass** examination to make sure your design is synthesizable. For Fairness, **please do logic synthesis with the provided scripts. Only two parts of the scripts can be modified**. One is at **line 17** in **0_readfile.tcl**, you can add all your hdl files here. The other is at **line 5** in **synthesis.tcl**, you can modify the clock timing constraint “TEST_CYCLE” here. **The timing slack of your synthesis result should not be negative.**

1. Functionality Score (5%)

PART1 CONV1_dw (pass testbench w/ TEST_CONV1_DW flag) : 1%

PART2 CONV1_dw + CONV1_pw (pass testbench w/ TEST_CONV1_PW flag): 1%

PART3 CONV1_dw + CONV1_pw + CONV2_dw + CONV2_pw
(pass testbench w/ TEST_CONV2_PW flag): 1%

PART4 CONV1_dw + CONV1_pw + CONV2_dw + CONV2_pw + CONV3 + POOL
(pass testbench w/ TEST_CONV3_POOL flag): 2%

Note: For each PART, your design need to pass all pattern (i.e. PAT_L=0, PAT_U=99) to get the corresponding full functionality score.

2. Basic Synthesis Score (6%)

(Only for those who finish whole design, i.e. PASS tb w/ TEST_CONV3_POOL, PAT_L=0, and PAT_U=99)

PI Grade	Synthesis PI	Basic Synthesis Score
A	$PI \leq 2.5 \times 10^9$	6
B	$2.5 \times 10^9 < PI \leq 5.0 \times 10^9$	4
C	$PI > 5.0 \times 10^9$	2

The **Performance index(PI)** for this assignment is defined as:

$$PI = A \times T \times C$$

A: Total cell area which is shown in report_area_Convnet_top.out

T: TEST_CYCLE (your setting of clock timing constraint when synthesis)

C: Average cycle count per pattern. It is shown on the terminal such as below when the whole simulation is passed and finished

```
Total cycle count = 1353101
Average cycle count per pattern = 13531
$finish called from file "test_top.v", line 1559.
$finish at simulation time      135310000
VCS Simulation Report
```

3. PI Ranking Score (4%)

(Only for those who achieve PI Grade A and submit the homework before the deadline)

As illustrated in Fig. 23, the PI Ranking Score is decided by your relative rank in the class.

0%-25%: 4	25%-50%: 3	50%-75%: 2	75%-100%: 1
PI Grade: A		PI Grade: B	PI Grade: C
			Others

Fig. 23 An illustration of PI ranking score

Bonus

Among works that achieve PI Grade A, **3%** will be given to the best work in terms of Performance index (PI), and **1%** will be given to the second-best work. (Only for those who submit the homework before the deadline)

Simulation

Simulation commands for each part are as below (in run_rtl.sh)

```
## part1: CONV1_DW ##
vcs -R +v2k -full64 -f sim.f -debug_acc -l vcs_part1.log \
+define+TEST_CONV1_DW+PAT_L=0+PAT_U=99 \
+define+FLAG_VERBOSE=0 \
+define+FLAG_SHOWNUM=0 \
+define+FLAG_DUMPWV=0

## part2: CONV1_DW + CONV1_PW ##
vcs -R +v2k -full64 -f sim.f -debug_acc -l vcs_part2.log \
+define+TEST_CONV1_PW+PAT_L=0+PAT_U=99 \
+define+FLAG_VERBOSE=0 \
+define+FLAG_SHOWNUM=0 \
+define+FLAG_DUMPWV=0

## part3: CONV1_DW + CONV1_PW + CONV2_DW + CONV2_PW ##
vcs -R +v2k -full64 -f sim.f -debug_acc -l vcs_part3.log \
+define+TEST_CONV2_PW+PAT_L=0+PAT_U=99 \
+define+FLAG_VERBOSE=0 \
+define+FLAG_SHOWNUM=0 \
+define+FLAG_DUMPWV=0

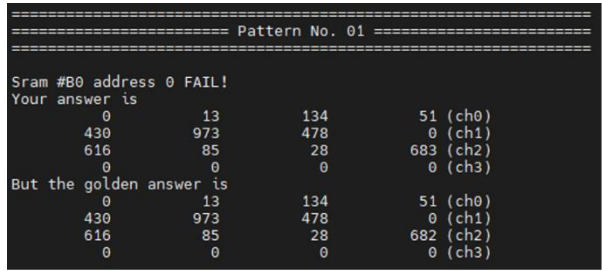
## part4: CONV1_DW + CONV1_PW + CONV2_DW + CONV2_PW + CONV3 + POOL ##
vcs -R +v2k -full64 -f sim.f -debug_acc -l vcs_part4.log \
+define+TEST_CONV3_POOL+PAT_L=0+PAT_U=99 \
+define+FLAG_VERBOSE=0 \
+define+FLAG_SHOWNUM=0 \
+define+FLAG_DUMPWV=0
```

For debugging convenience, we provide a few flags to meet specific simulation requirements.

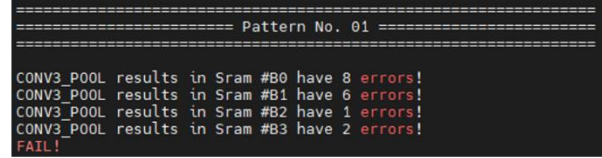
1. TEST_*: Specify the layer you want to test (e.g. TEST_CONV1_DW, TEST_CONV1_PW, TEST_CONV2_DW, TEST_CONV2_PW, TEST_CONV3_POOL)

2. PAT_L and PAT_U: Specify specific range of patterns.
3. FLAG_VERBOSE: Show the detailed simulation reports or not

As shown in **Fig. 24**, when you set FLAG_VERBOSE to 1, your wrong answers and the corresponding golden answers would be printed in the terminal. On the other hand, when you set FLAG_VERBOSE to 0, only brief functionality test results will be provided.



FLAG_VERBOSE=1



FLAG_VERBOSE=0

Fig. 24 Introduction of FLAG_VERBOSE

4. FLAG_SHOWNUM: Display the digit classification information or not
5. FLAG_DUMPWV: Dump the fsdb waveform or not

Deliverable

1. Synthesizable Verilog

- Convnet_top.v
- All other your own defined RTL files (*.v) (if any)

2. Command-line argument file

- sim.f (should include all the RTL files used in your design)

3. Spyglass rpt

- spyglass.rpt

4. Synthesis script and results

- 0_readfile.tcl
- synthesis.tcl
- da.log
- report_time_Convnet_top.out
- report_area_Convnet_top.out
- Convnet_top_syn.v
- Convnet_top_syn.sdf

5. Text file

- misc.txt

Summarize your Grade and PI, including each value of A, T, and C of your submitted design. A template is provided in the released package.

Directory	Filename
HW5_{studentID}/	misc.txt
	spyglass.rpt
HW5_{studentID}/hdl/	Convnet_top.v
	All other your own defined RTL files (*.v) (if any)
HW5_{studentID}/sim/	sim.f
HW5_{studentID}/syn/	0_readfile.tcl
	synthesis.tcl
	da.log
	report_time_Convnet_top.out
	report_area_Convnet_top.out
	Convnet_top_syn.v
	Convnet_top_syn.sdf

Note:

If your studentID is m123456789, HW5_{studentID} denotes HW5_123456789.

Compress your whole folder HW5_{studentID} into **HW5_{studentID}.zip** and submit to eeclass

Wrong file delivery or wrong file organization will get 1% punishment.

Appendix

■ RTL I/O Port Name Definition

Port Name	I/O	Bitwidth	Description
clk	I	1	Clock signal
srst_n	I	1	Synchronous reset (active low)
enable	I	1	1 : The unshuffled image is ready in SRAM A (one-cycle pulse)
valid	O	1	Finish writing feature maps to corresponding SRAM group
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/ 5	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/ 5	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	7	Read address for SRAM_bias
sram_rdata_bias	I	8	Read data from SRAM_bias

■ Description of Transfer Files

Directory	Filename	Description
EE4292_HW5_2022/	misc.txt	txt file for recording grade and PI
EE4292_HW5_2022/bmp/	test *.bmp	Mnist hand write datasets (test image 0~99)
EE4292_HW5_2022/hdl/	Convnet_top.v	Top module
EE4292_HW5_2022/sim/	feature_map/*.npy	Python file for debugging
	golden/*.dat	Golden results of feature maps
	param/*.dat	Trained weights and biases of each layer
	sram_model/*.v	SRAM behavior model
	check_feature.py	Program for you to see all layer outputs
	sim.f	File list for RTL simulation
	test_top.v	Testbench
	run_rtl.sh	script for pre-sim
	clean.sh	script for space cleaning
EE4292_HW5_2022/syn/	.synopsys_dc.setup	DC environment file
	*.tcl	tcl file for DC scripts
	syn.sh	script for synthesis

■ Notes

- For debugging, we have provided *check_feature.py* for you to check all layers' output feature maps. **Fig. 24** shows how to run this python file. First, log in EE workstation wsXX, and then type *python check_feature.py*. After entering the pattern, layer, and channel number, it will print the corresponding results. (If it doesn't work, try *python2 check_feature.py*.)

```

$ python check_feature.py
Enter the pattern you want to show (0 ~ 99):
0
Enter the layer you want to show:
0 for unshuffle,
1 for conv1_dw,
2 for conv1_pw,
3 for conv2_dw,
4 for conv2_pw,
5 for conv3_pool
0
Enter the channel you want to show (0 ~ 3):
0
[[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0 127 127 127 127 127 127 127 127 0  0  0]
 [ 0  0  0  0  0  0  0  66  67  67  21 127 0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  127  83 0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  59 127 0  0  0]
 [ 0  0  0  0  0  0  0  0  0 127  58  0  0  0  0]
 [ 0  0  0  0  0  0  0  75 127  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  3 127  35  0  0  0  0  0]
 [ 0  0  0  0  0  0  0 127 115  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  61 127  52  0  0  0  0  0  0]
 [ 0  0  0  0  0  0 121 127  0  0  0  0  0  0  0]]

```

Fig.24 An example of how to run *check_feature.py*.

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