National Tsing Hua University
Department of Electrical Engineering
EE4292 IC Design Laboratory (積體電路設計實驗)
Fall 2018

Homework Assignment #5 **(15%)**

CNN: LeNet inference for digit recognition

Assigned on Nov 22, 2018
Due by Dec 13, 2018

- **Introduction**

LeNet is one of the earliest convolutional neural networks to promote the development of deep learning. After many successful improvement, Yann LeCun named it LeNet5 in 1988. At that time, LeNet architecture was primarily used for digit recognition tasks such as reading postal codes, numbers, and more. In HW5, we will implement modified LeNet as shown in Fig. 1, and we only focus on the implementation of CONV1_1, CONV1_2 and POOL1. The other parts (CONV2, POOL2, FT, FC1 and FC2) are provide in testbench by TA.

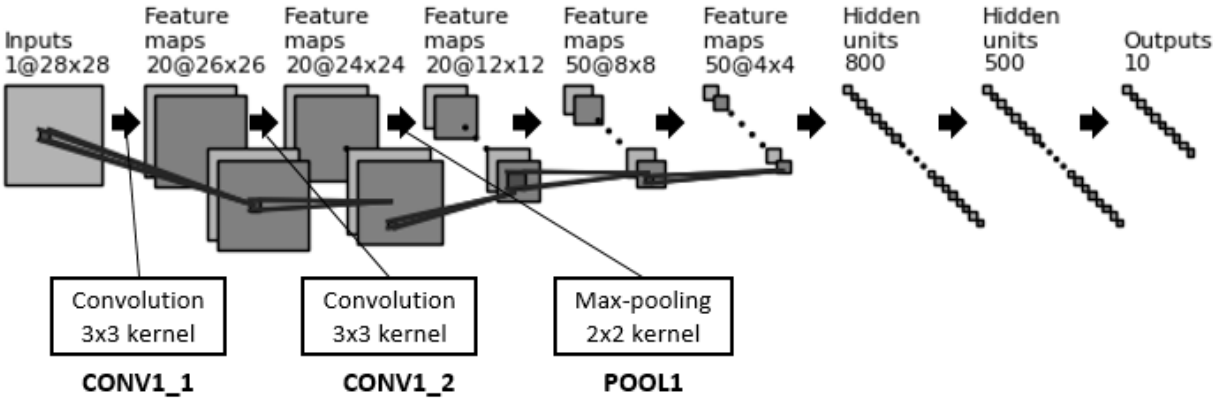


Fig. 1 *Architecture of modified LeNet[1]. In this assignment, you only need to implement the CONV1_1, CONV1_2, POOL1 layer. Rest of the layers are completed in the testbench for your reference only.*

- **Assignment description**

For this assignment, we will provide two SRAM group A and B (each group has four SRAM banks) for you to save the feature maps. First, you will load into an image which its size is **28x28** from the testbench, and save the input data to SRAM group A. Besides, the image will be loaded in **raster scan order (z-scan)** as shown in Fig. 2, and **a pixel per cycle.** After loaded input image, do the 2D convolution to get the CONV1_1's feature maps, and write the CONV1_1's feature maps to SRAM B. Then, do the 3D convolution to get the CONV1_2's feature map, and immediately do the pooling to get the POOL1's result. Finally, write the POOL1's result back to SRAM group A, and the testbench will test your answer. The whole data flow as shown in Fig. 3.

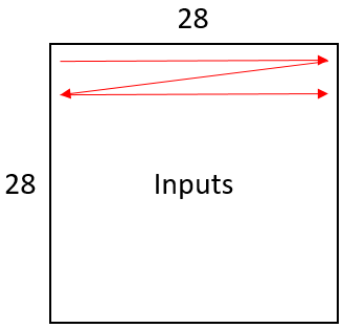


Fig. 2 Z-scan order for loading an image.

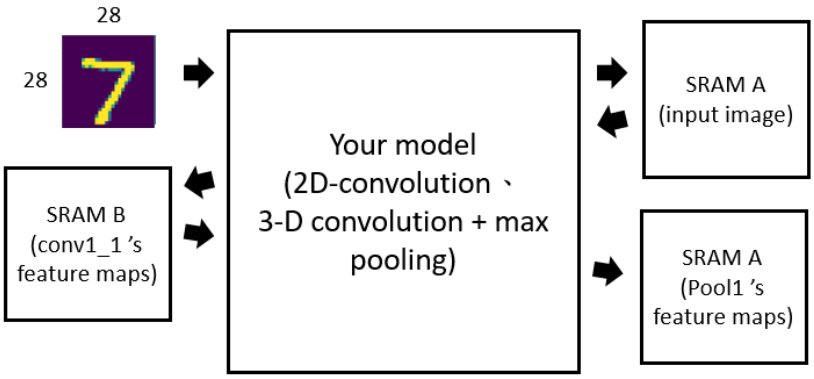


Fig. 3 Data flow in this assignment.

- **Algorithm**

In this section, we'll introduce the operations in CONV1_1, CONV1_2, and POOL1. If you're familiar with convolution and max-pooling, you can skip this section.

➢ **CONV1_1 Layer**

In CONV1_1 layer, you need to do 2D-convolution on the input feature map 20 times, each time using a different 2D kernel. So there will be 20 kernels, the size of each kernel is 3x3 as shown in Fig. 4. The convolution operation in this assignment is performed without padding. An example of a 5x5 image convolved by 3x3 kernel is shown in Fig. 5 to help you understand the concept of non-padding convolution. After the convolution is done, you will need to add another value called bias to every point in the result to get the final output feature map. Repeat this process 20 times, each time with different kernel and bias, will complete the operations in CONV1_1. The operations in CONV1_2 is almost the same except that you need to do 3D convolution instead of 2D convolution.

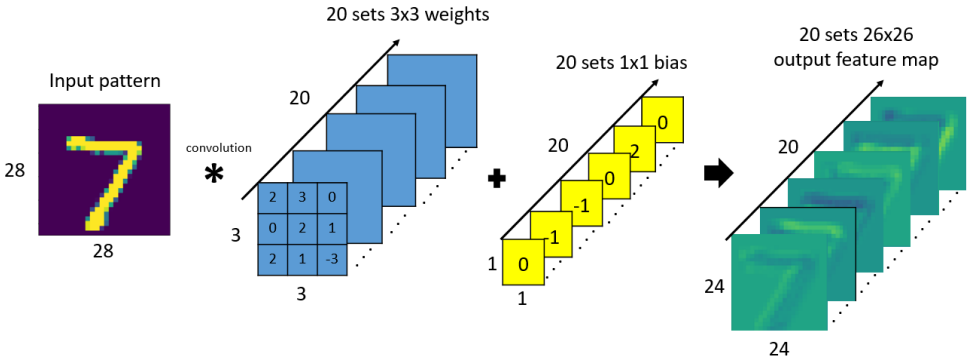


Fig. 4 Detail for CONV1_1 layer, a 2D image (28x28) is convolved with 20 different 2D kernels to generate 20 output feature maps. Note that after each 2D convolution, a bias will be added to the convolution result (at each point) to get the final output feature map.

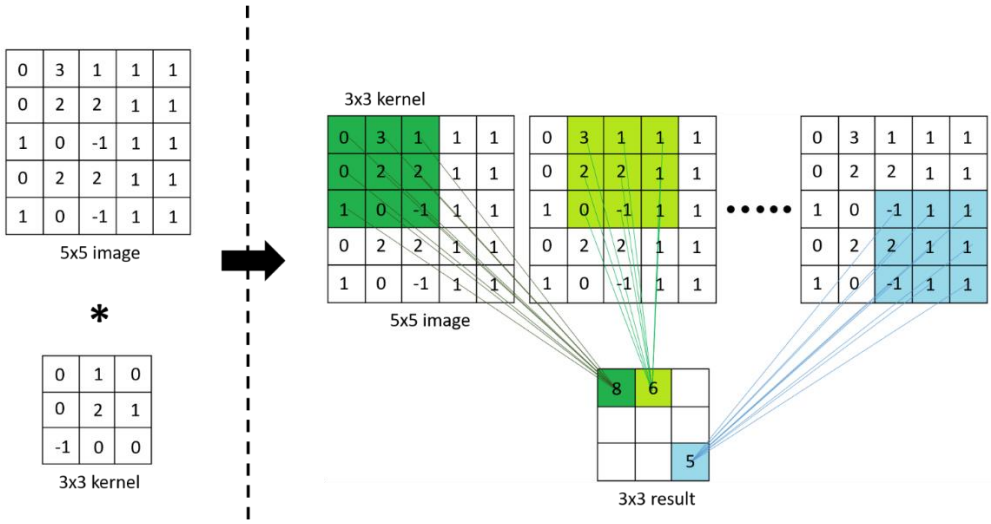


Fig. 5 *Example showing a 5x5 image convolved by a 3x3 kernel. During convolution, we move the kernel completely inside the image (thus no padding is need) and take the value of feature map inside the window to perform dot product with the kernel. Note that if the size of image and kernel is NxN and MxM, the result size of the convolution will be N+M-1. So in CONV1 you will found that the output feature map size is 28+3-1=26.*

- **CONV1_2 Layer**

In CONV1_2 layer, you need to do 3D convolution on the input feature 20 times, each time using a different 3D kernel. So you will need 20 sets 3D kernels to generate 20 output feature maps. Because the size of input feature maps is 20x26x26, the size of each kernel is 20x3x3 as shown in Fig. 6. An example of 3D convolution using weight No.0 is shown in Fig. 7. After the 3D convolution, a bias is added to every point of this 3D convolution result (1x24x24) to get the output feature map. The convolution operation in CONV1_2 layer is also performed without padding. After repeating the 3D convolution 20 times with 20 different 3D kernels, we will get 20 sets 24x24 CONV1_2 layer's output feature maps.

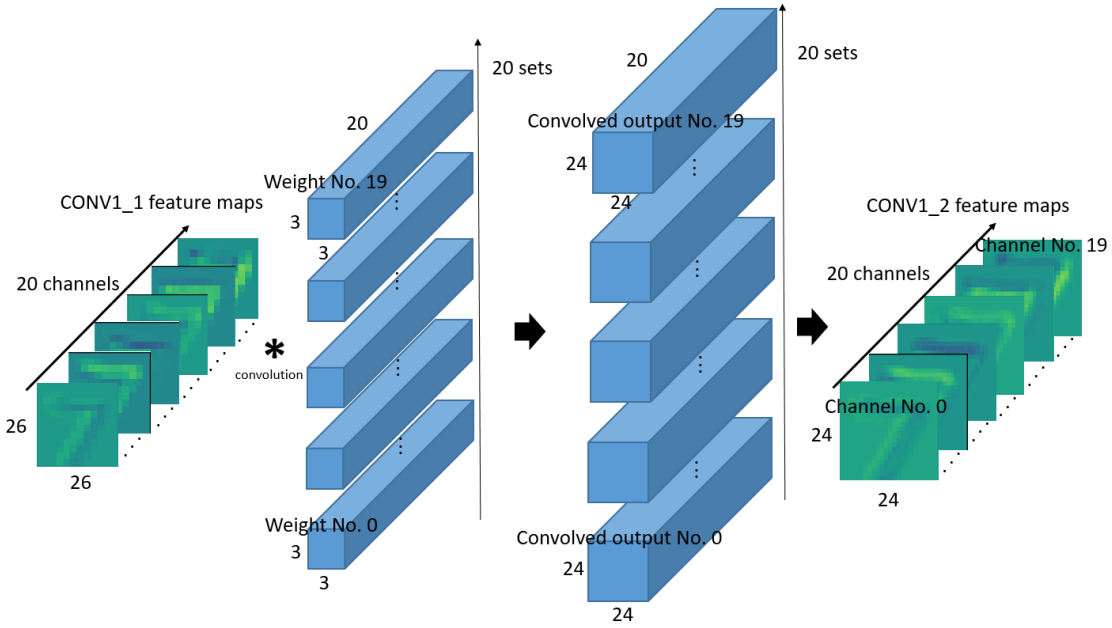


Fig. 6 *Example for CONV1_2 layer, CONV1_1 feature maps (20x26x26) are convolved with 20 different sets of 3D kernels to generate 20 output feature maps.*

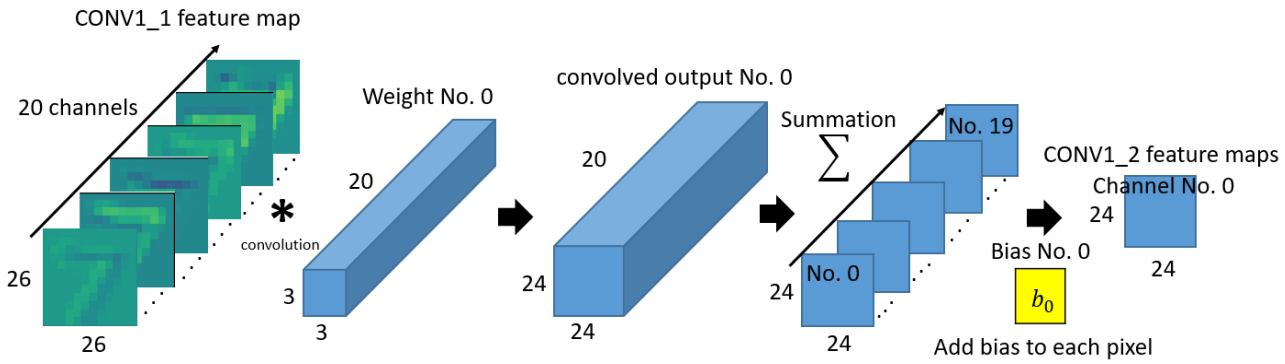


Fig. 7 *Example for 3D convolution with weight No. 0. CONV1_1 feature maps (20x26x26) are convolved with 3D kernels (20x3x3) without padding to generate convolved output (1x24x24). Because they have the same number of channels, you can first perform 2D convolution channel by channel, and then sum up those results along channel-direction to get the 3D convolution result. Note that we have to add a bias to the 3D convolution result to get the output feature map just as what we did in CONV1_1.*

- **POOL1 Layer**

  Pooling is another important concept in CNN, which is actually a form of down sampling. There are many different forms of nonlinear pooling functions, and max-pooling is the most common method which select the biggest value inside the pooling kernel. Here we will perform 2x2 max-pooling. An example is shown in Fig. 8. After max-pooling, we can get 20 sets 12x12 feature maps in POOL1 layer as shown in Fig. 9.

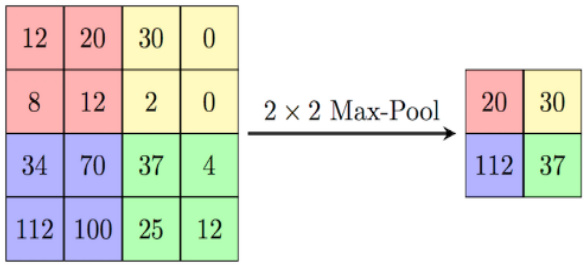


Fig. 8 *Example of 2x2 Max-Pooling*

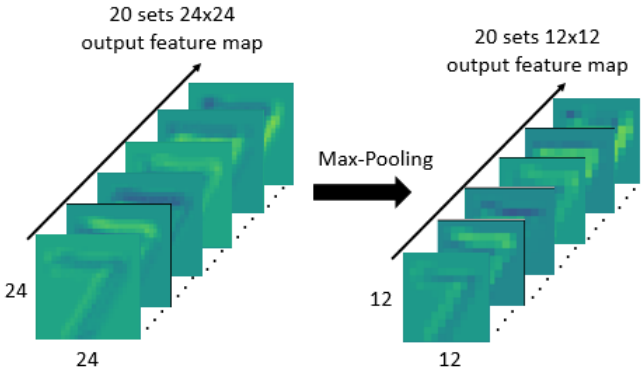


Fig. 9 *Detail for POOL1 layer*

- **Hardware design**

According to the assignment description, you have to design a circuit which can do both the 2D-convolution and 3D-convolution. Besides, we recommend you using a big FSM separated into 3 small FSM "**loading image**", "**CONV1_1**" and "**CONV1_2 + Max pooling**", and the block diagram is supposed to be like **Fig. 10**. Here we will provide a better way to do the CONV1_2 + Max pooling in the following. In consideration of cycle counts and SRAM size, we don't want to do pooling until the CONV1_2 layer was done, because it will spend too much area keeping the temporary result. In order to get one pooling result immediately, we need to generate 4 neighboring convolution results each cycle, and that would require 4x4 = 16 pixels from input feature map. An example of this is shown in **Fig 11**.

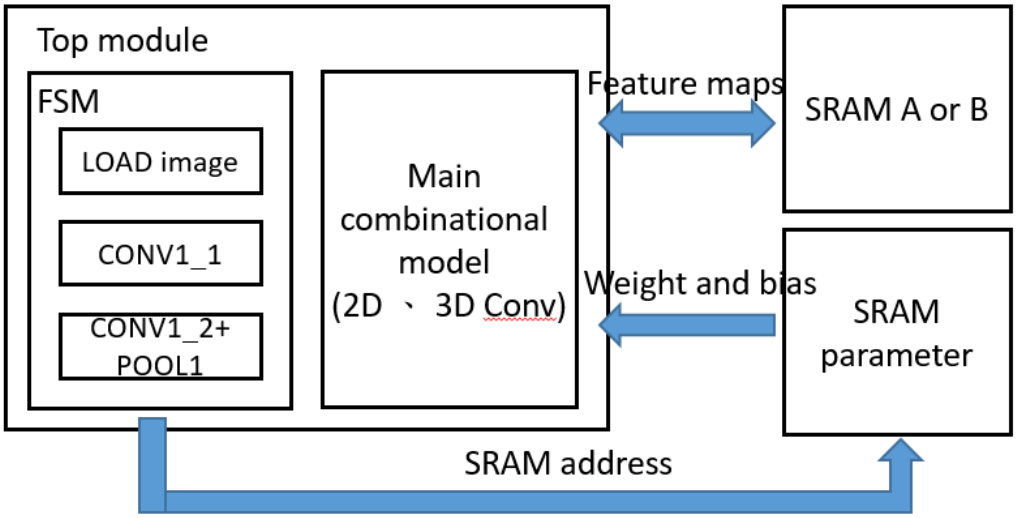


Fig. 10 *System block diagram for this assignment.*

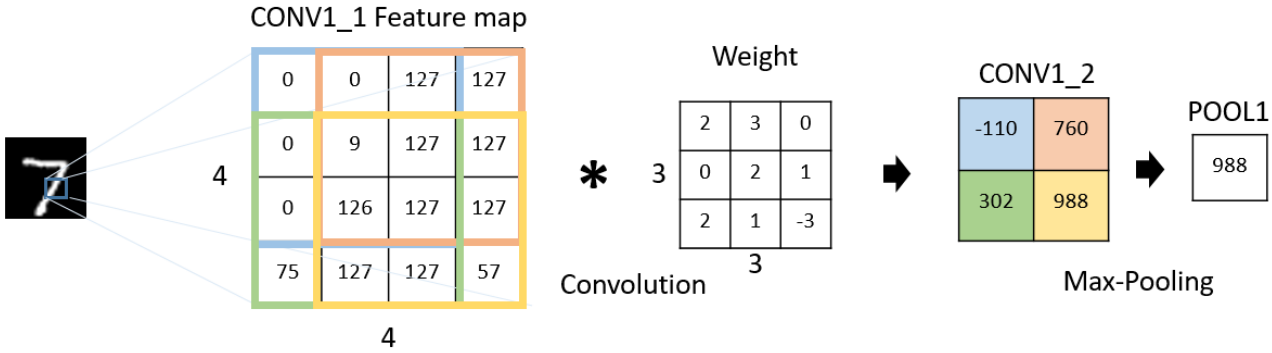


Fig. 11 *Example of generating one pooling result in just one cycle. First we need to read 4x4 pixels in the CONV1_1 feature map. Convolve them with a 3x3 kernel will generate 4 neighboring results. These neighboring results would be sufficient to perform 2x2 max-pooling.*

In order to get 16 pixels in one cycle, **we store the feature maps in 4 SRAM banks**, **each bank stores 4 pixels per address.** So in each cycle, by feeding 4 addresses to 4 banks, total 4x4=36 pixels can be reached from SRAM. An example of sub-bank data storage is shown in **Fig. 11.**

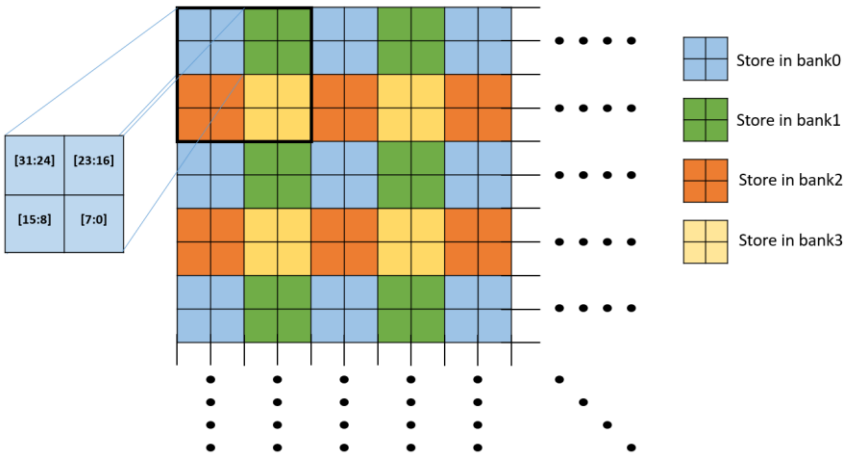


Fig .11 *Store the feature maps into 4 sub SRAM banks, blue for example is stored in SRAM_BANK0. By feeding 4 addresses to 4 banks respectively, one can get 4 data (4 pixels per data) from SRAM in one cycle.*

- **Problem definition**

Part 1. Loading input image to SRAM group A(2%)

In this part, you need to use 4-bit bytemask to write data in SRAM is shown in **Fig. 12**. Besides, you only need to finish loading image according to the specific way as shown in **Fig. 13**. You can use **"busy"** to control whether the new input data can be loaded. When you set **"valid"** to **"1",** testbench will check your answer in SRAM immediately. By the way, you don't need to do the synthesis, and you just need to pass the simulation for pattern No. 0.

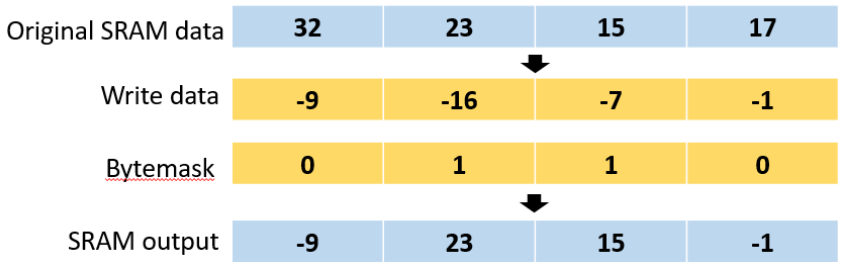


Fig .12 *SRAM behavior when using 4-bit bytemask to write data into SRAM. Here you should be aware of that the bytemask is* ***"active low"****.*

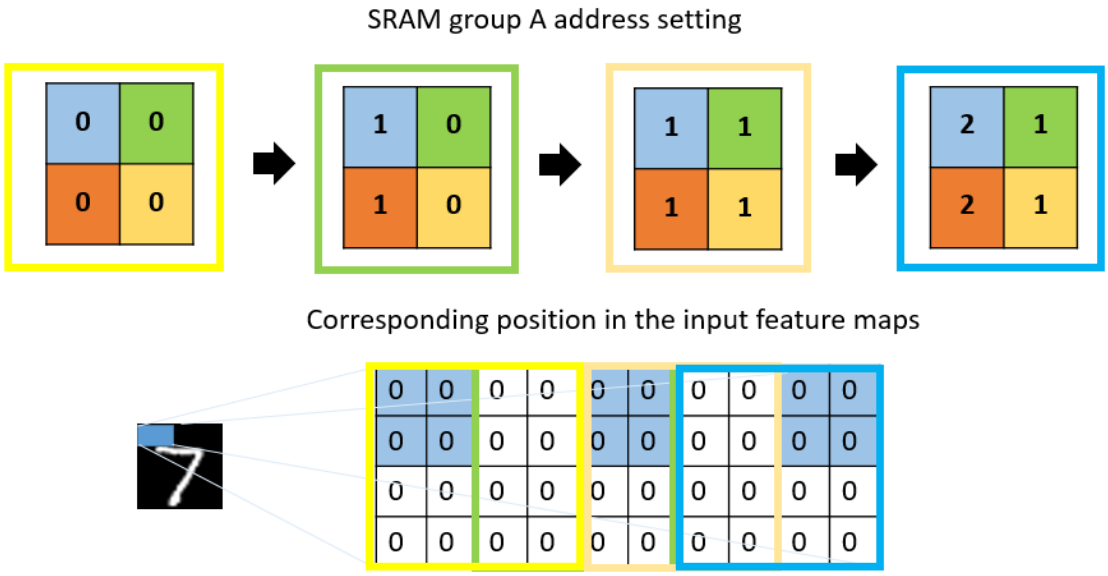


Fig .13 *Store the feature maps into 4 sub SRAM banks, blue part for example is stored in SRAM group A bank 0.*

You need to complete the following missions in PART1:

1. RTL coding and simulation (1%).
2. Use nLint to show the synthesizability (1%).

PART2. Design a circuit to implement 2D、3D convolution and Max pooling (13%)

The detail for implementing PART2

The following steps for implementing PART2 will be required for your FSM:

1. Use "busy" to control when to load image;
2. Implement 2D convolution operation;
3. Quantize the CONV1_1 layer's output to **8 bit,** and do the **ReLU;**

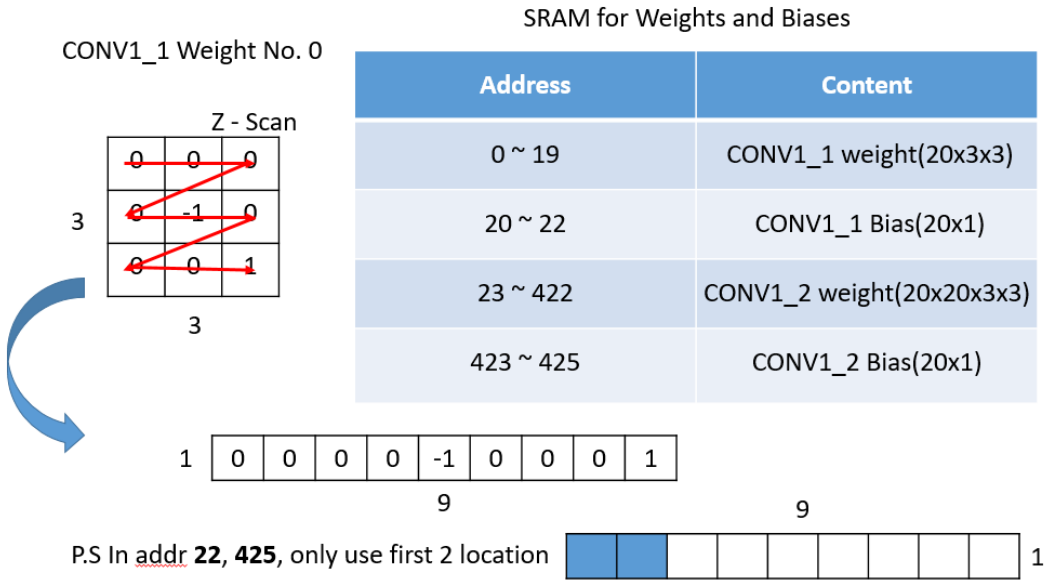
4. Implement 3D convolution operation and Max-pooling;
5. Quantize POOL1 layer's output to **8 bit,** and do the **ReLU;**
6. Write the POOL1 layer's output to **SRAM A;**

■ **Details of Step 1.**

In the testbench *test_top.v*, the way of loading image is the same as the PART1. In PART2, you can choose use **buffers** or **SRAMs** to save the data. It's up to you.

■ **Details of Step 2 and Step 4**

In the testbench *test_top.v*, another SRAM behavior model *sram_1024x32b.v* is used to store weights and biases. Because we want to read 3x3 kernel in one cycle, so we reshape 3x3 kernel into 9x1 and load it into one SRAM address. Because the bit width of weights and bias is 4, there are 36 bits per SRAM address. **Fig.11** shows you the location of these weights and biases.



SRAM for Weights and Biases

Address	Content
0 ~ 19	CONV1_1 weight(20x3x3)
20 ~ 22	CONV1_1 Bias(20x1)
23 ~ 422	CONV1_2 weight(20x20x3x3)
423 ~ 425	CONV1_2 Bias(20x1)

Fig.11 the location of Weights and Biases in SRAM_1024x36b

■ **Details of Step 4.**

We don't recommend you to write CONV1_2's results into SRAM for buffering, otherwise the dataflow will be complicated. The better way is to accumulate your temporal convolved output by changing your weights form channel-0 to channel-19. After accumulation, do max-pooling to get POOL1's feature maps.

■ **Details of Step 3 and Step 5.**

32-bit floating point is commonly used on GPUs and CPUs, and it has a wide dynamic range which is beneficial for accuracy. However, it requires high energy and area costs. So, we use fixed point to represent the fractional number instead of using 32-bit floating point. Details are given below:

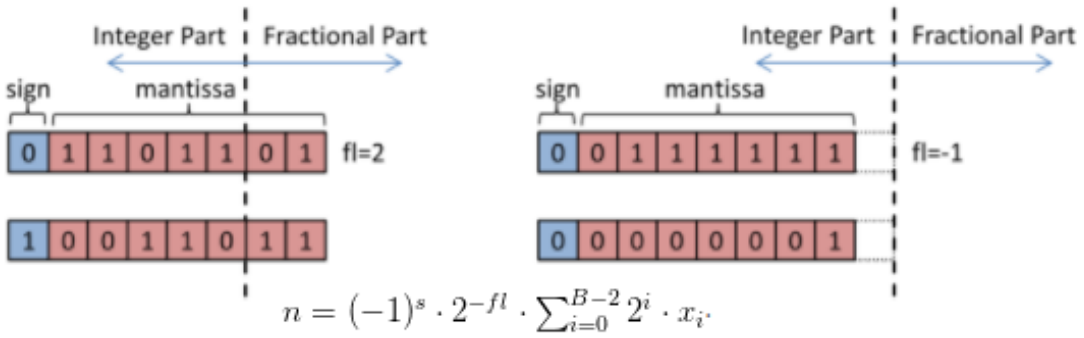


Fig. 16 ***B*** *represent the bit-width,* ***s*** *is the sign bit,* ***fl*** *is the fractional length, and* ***x*** *is the mantissa bits.*

	Bit width	Fractional length(precision)
Input feature map	8	8
Output feature map	8	6
Weight	4	3
Bias	4	5

Table. 1 *the bw and fl of data in CONV1_1 layer.*

	Bit width	Fractional length(precision)
Input feature map	8	6
Output feature map	8	4
Weight	4	5
Bias	4	7

Table. 2 *the bw and fl of data in CONV1_2 layer.*

Taking CONV1_2 layer for example, we can find that the fractional length of multiplied results from input feature maps and weights is 11 bit in **Table.2**. So when you add a bias to your accumulated output, in order to align the fractional length of accumulated output with bias, you need to shift bias 4 bit (11bit – 7bit). Besides, we need to do the rounding for better precision as shown in Fig. 17. Then, shifting the fractional length from 11 bit to 4 bit to get the CONV1_2 layer's result. Finally, add the **ReLU function** to introduce non-linearity in this model.

$$\text{ReLU function } f(x) = x^+ = \max(0, x)$$

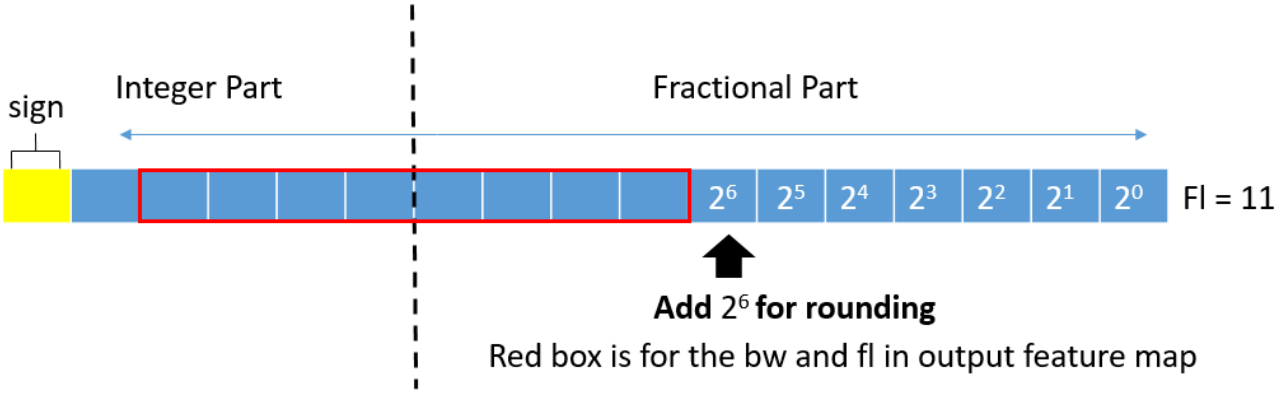


Fig. 17 *The illustration of doing rounding. No matter the sign bit is 0 or 1, we all add 1 to rounding bit.*

The pseudo code for quantizing will help you get the right answer in CONV1_2's layer:

```
accumulated_output = accumulated_output + (bias << 4) + 2^6;
accumulated_output = accumulated_output >>> 7;
if      (accumulated_output >= 127)     q_output = 127;
else if (accumulated_output < 0)        q_output = 0;     add the ReLU function
else                                    q_output = accumulated_output [7:0];
```

■ **Details of Step 6.**

There are 20x12x12(totally 2880 pixels) output feature maps in the POOL1 Layer, and we hope you can write answer to **SRAM a0 one by one, and four pixels per address.** So, it totally will use address 0~719 to store POOL1 layer's result. Besides, we apply "**Bytemask**" here to help us write 8-bit data into 32bit of SRAM.

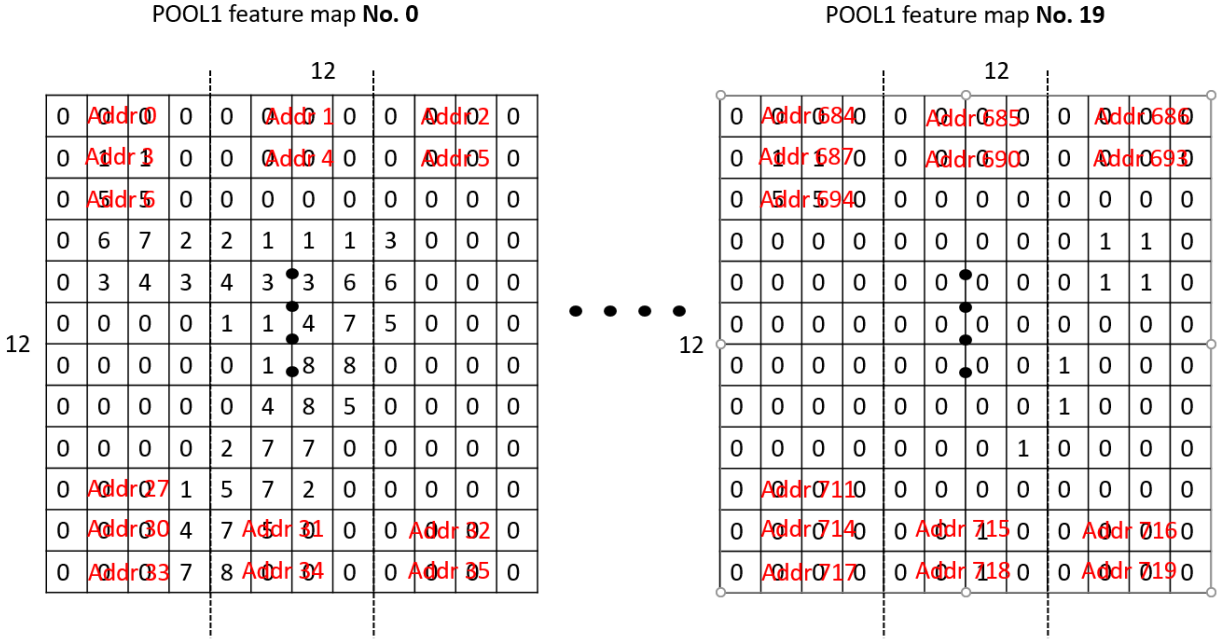
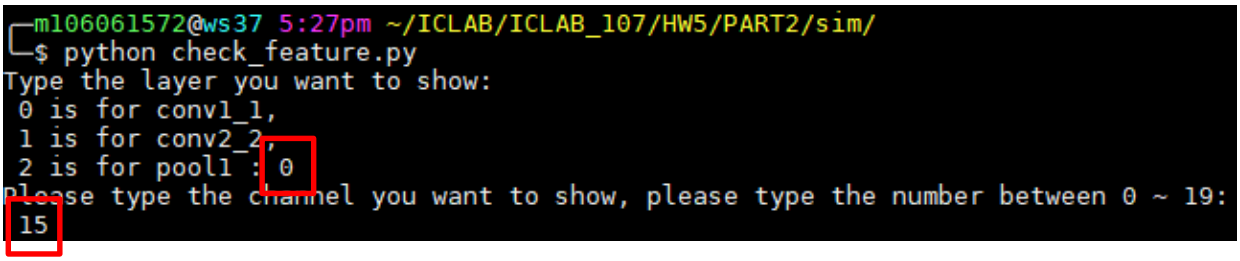


Fig. 16 The example of using "bytemask" to write data in SRAM.

Note:

For debugging, you can run "**check_feature.py**" to display the selected layer and channel. For example, if you want to show the Conv1_1 layer's channel No.15 for the pattern **No. 0,** you can use the following command:



You need to complete the following missions:

1. RTL coding and simulation (4%).
2. Use nLint to show the synthesizability (1%).
3. 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. 2.1 for 2.1 ns). Your timing slack should not be negative. The grading will be based on your performance index (defined later):

$$\text{Grade point: } \begin{cases} 3\% & if \ \ \boldsymbol{PI} > 8\times10^9 \\ 6\% & if \ \ 8\times10^9 \geq \boldsymbol{PI} > 6\times10^9, \\ 8\% & if \ \ \boldsymbol{PI} \leq 6\times10^9 \end{cases}$$

Definition of performance index:

A: Total area (shown in report/report_area_conv2_top.out, e.g. Total area: 37047)
T: TEST_CYCLE (your timing constraint, e.g. 2.5 for a nice implementation)
C: The cycle count you need to complete the simulation of *test_top.v*, which is shown on the monitor when the simulation is finished, for example:

Congratulations! Simulation from pattern no 0000 to 0000 is successfully passed!
Total cycle count = 61788!

✧ **Performance index = A*T*C** (e.g. $5.72*10^9$ for the above example)

- **Deliverable**

1. Synthesizable Verilog functional module *top.v* and all other RTL codes for your own defined modules (if any).
2. Create a file named *hdl.f*, which contains all the RTL files (*.v) you use in this assignment.
3. nLint report file for PART1 and PART2 (***nLint*** rule set).
4. A text file ***misc.txt*** summarizing performance index and your A, T, and C as well.

File Organization

- **PART1**

Directory	Filename	Description
hdl	top.v	RTL code for loading image to SRAM A.
sim	test_top.v	Testbench for loading image to SRAM A.
	sram_model/sram_512x36b.v	SRAM behavior model
	sram_model/sram_1024x32b.v	SRAM behavior model
	parameter/*.dat	Trained weights and Biases of each layer
	golden/*.dat	Golden results

- **PART2**

Directory	Filename	Description
hdl	top.v	RTL code for CONV1_1, CONV1_2 and POOL1 in LeNet (modify it directly for this assignment)
	hdl.f	All the RTL files (*.v) you use in this assignment.
sim	test_top.v	Testbench for LeNet inference
	sram_model/sram_512x36b.v	SRAM behavior model
	sram_model/sram_1024x32b.v	SRAM behavior model
	parameter/*.dat	Trained weights and Biases of each layer
	golden/*.dat	Golden results
syn	*.tcl	tcl file for DC scripts
	run_dc.bat	Batch file for running synthesis

P.S Don't put the *.bmp in the folder which you want to submit!!!!

- **Reference**

[1] Lenet-5 implement ,

[2] Wikipedia, https://en.wikipedia.org/wiki/Convolutional_neural_network

[3] Ristretto, http://lepsucd.com/?page_id=621