

PROJECT1:

Cat and dog classification CNN model

Team1: DO QUANG HUY, 강덕현, 권용환

1. Python

1) 설계 목표

모델을 설계하면서 가장 중요하게 생각한 것은 복잡도를 줄이는 것이었다. 많은 CNN예제에서는 다양한 optimizer, batch normalization 등을 이용하고, kernel size도 변화시키면서 모델의 성능을 높이곤 한다. 그러나, 우리는 hardware implementation을 목표로 하고 있기 때문에 최대한 복잡도를 줄이고, 파라미터 수도 줄여야 한다. 이를 위해서 layer의 숫자를 줄이는 것부터 시작하였다. CNN은 기본적으로 input layer에서 첫번째 hidden layer로 들어갈 때, 2D convolution을 하게 된다. 2D convolution layer를 몇 개나 구성하든 관계없이 높은 accuracy를 내기 위해선 반드시 FC(Fully-Connected)layer가 동반된다. FC는 N장의 이미지의 모든 pixel에 대한 correlation을 구하여 숨겨진 hidden feature를 끌어내는 과정이다. 2D convolution또한 correlation을 구하는 과정으로 볼 수는 있으나 일반적으로 layer를 얇게 가져가는 경우 FC의 역할이 중요하게 된다. 따라서 우리는 얇은 layer를 만들면서 2D convolution의 경우 filter개수를 줄이고, 첫번째 FC를 비교적 적당한 수준으로 만들어 얇은 layer에서 모델을 구현할 것이다. 오히려 layer가 지나치게 깊어지게 되면 2D convolution 과정에서 Weight에 대한 correlation분포가 특정 값에 치우치게 되는 covariance shift현상이 발생하게 되고, 이 경우 batch normalization등에 대한 추가적인 모듈이 요구된다.

앞서 언급한 간단한 모델의 설계를 위해 2D convolution의 kernel size는 3x3, stride는 1로 하여 가장 보편적이고, 간단한 형태의 kernel을 이용할 것이다. 또한, 모든 hidden layer에서는 ReLu activation을 사용하여 local minima에 빠지는 현상을 피하고자 한다. Output layer는 sigmoid로 구현하여 hardware에서도 비교적 쉽게 모델링할 수 있도록 한다. Optimizer는 ADAM, MSE방식을 선택했다. Optimizer또한 직접 구현하는 것이 Software와 Hardware level의 간극을 줄이는데 중요한 역할을 할 수 있으나 현재는 optimizer를 직접 구현하는 수준의 작업은 어렵다고 판단되어 TensorFlow의 optimizer를 그대로 사용하고자 한다. 이번 모델의 모든 코드는 TensorFlow를 이용하여 구현되었다.

결과적으로, 우리는 input layer, 3개의 hidden layer, output layer로 모델을 설계하였다. 데이터 전 처리 과정에서 빠른 학습을 위해 image를 8bit gray scale로 한 후에 0~1.0으로 normalization하였는데, 0과 1로 binary quantization을 한 결과 더 좋은 accuracy를 보였다. Binary quantization이 진행된 데이터 셋은 image size를 58x58까지 resize해도 95~99%의 좋은 accuracy를 보여주었다.

2) 코드 분석

```
import numpy as np
import pandas as pd
import cv2
import matplotlib.pyplot as plt
import random
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import tensorflow as tf
import zipfile
from turtle import shape
from tensorflow.python.keras import Model
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Flatten, Dropout, Activation, Conv2D, MaxPooling2D
##### Extract Zip file #####
with zipfile.ZipFile("C:/Users/user/Desktop/PROJECT1/train.zip", "r") as z:
    z.extractall(".")
with zipfile.ZipFile("C:/Users/user/Desktop/PROJECT1/test1.zip", "r") as z:
    z.extractall(".')
```

Fig 1

사용된 패키지는 numpy, pandas, cv2, matplotlib, random, os, tensorflow, zipfile, shape이다. Zipfile은 Kaggle에서 다운로드 한 zip파일의 압축을 풀기 위해 사용되었다.

```
##### Train and Pre-processing #####
main_dir = "C:/Users/user/Desktop/PROJECT1/"
train_dir = "train/"
path = os.path.join(main_dir, train_dir)

X = []
y = []
convert = lambda category : int(category == 'dog')
def create_train_data(path):
    for p in os.listdir(path):
        category = p.split(".")[0]
        category = convert(category)
        img_array = cv2.imread(os.path.join(path, p), cv2.IMREAD_GRAYSCALE)
        new_img_array = cv2.resize(img_array, dsize=(58,58))
        X.append(new_img_array)
        y.append(category)

create_train_data(path)
X = np.array(X).reshape(-1,58,58,1)
y = np.array(y)
# Binary Quantize Input data
X = X/128
X = X.astype(int)
# Quantized Random Binary Image Plot
index = random.randint(0,25000)
plt.imshow(X[index])
```

Fig 2

데이터를 전 처리하는 역할을 한다. 우리가 사용하는 데이터 셋은 training data, test data로 구분되어 있는데 training data의 경우 dog1, dog2, cat1, cat2 등으로 naming이 되어 있고, test data의 경우 1, 2, 3 등으로 naming이 되어 있다. 따라서 training data의 경우 디렉토리 내의 jpg를 순서대로 읽으면서 dog이라는 이름이 있는 경우 1이라는 label을 붙일 수 있다. $X = X/128$ 을 이용하여 training data는 binary화 하였다. 처음부터 의도된 작업은 아니었고, binary화 한 후에 이미지 사이지를 100x100에

서 58x58까지 줄여도 높은 accuracy가 보장되어 위와 같이 binary quantization을 하였다.

```
##### Model #####
# input layer
input_layer = tf.keras.layers.InputLayer(input_shape=X.shape[1:])
# layer1
layer_conv1 = tf.keras.layers.Conv2D(16,(3,3),activation='relu')
pool1 = tf.keras.layers.MaxPool2D(pool_size=(2,2))
# layer2
layer_conv2 = tf.keras.layers.Conv2D(32,(3,3),activation='relu')
pool2 = tf.keras.layers.MaxPool2D(pool_size=(2,2))
# flatten layer
layer_flatten = tf.keras.layers.Flatten()
# layer3
layer_dense = tf.keras.layers.Dense(64,activation='relu')
# output layer
output_layer = tf.keras.layers.Dense(1,activation='sigmoid')

model = tf.keras.Sequential([
    input_layer,
    layer_conv1,
    pool1,
    layer_conv2,
    pool2,
    layer_flatten,
    layer_dense,
    output_layer
])
model.compile(optimizer='Adam', loss = 'mse', metrics = ['accuracy'])
model.fit(X,y,epochs=10,batch_size=32,validation_split=0.2)
model.summary()
```

Fig 3

구현한 모델이다.

- Input layer
- Conv2D layer, 3x3filter #16, activation relu
- 2x2 Max pooling
- Conv2D layer, 3x3 filter #32, activation relu
- 2x2 Max pooling
- Flatten
- Dense (Fully Connected) layer, neurons #64, activation relu
- Output layer, neurons #1, activation sigmoid

위와 같이 구성되었으며, 첫번째 FC의 뉴런 개수에 따라서 모델의 성능이 크게 바뀌는 것을 알 수 있었다. Training을 위해서 사용한 optimizer는 ADAM, loss는 MSE방식으로 구하였다. 검증을 위해 20%에 해당하는 5000장의 data를 validation set으로 하였다. Training의 경우 over-fitting이 나는 것을 방지하기 위해 batch size를 32로 하여 32개의 data가 입력될 때마다 한 번씩 weight를 update 하였다.

```
##### Find Maximum value of Activations #####
intermediate_layer_model = tf.keras.Model(inputs=model.input, outputs = model.layers[5].output)
intermediate_output = intermediate_layer_model(X) # tensor
intermediate_output_numpy = intermediate_output.numpy()
maximum_firstdense = 0
for inter in intermediate_output_numpy:
    if maximum_firstdense < max(inter):
        maximum_firstdense = max(inter)
```

Fig 4

위의 코드를 이용해서 layer[5](Dense)의 출력의 최대값을 구한다. 처음에는 모든 layer의 출력 값을 하나의 tensor array에 저장하였는데 이 경우 최대값을 찾기 위해 매우 많은 for문의 중첩이 요구되고 for loop의 경우 series iteration이기 때문인지 GPU가 아닌 CPU에서 작업을 수행하여 iteration limit에 걸려 코드가 동작을 멈추는 현상이 계속 발생하였다. 따라서 하나의 layer씩 확인을 했고, 당연하게도 하나의 neuron에 5408개의 weight summation이 요구되는 Dense layer에서 가장 큰 activation value를 구할 수 있었다. Training을 할 때마다 조금씩 변하기는 하지만 보통 2D Convolution layer는 2~3대의 값이 maximum으로 추출되는데, Dense layer는 평균적으로 25정도의 값을 maximum value로 출력한다. 따라서 Dense layer를 maximum value를 추출하기 위한 layer로 fix하여 사용하였다.

```
##### Quantization #####
Qa = 255/maximum_firstdense # Quantization facotr for Activation - unsigned 8bits
Qw = 127/max_weight # Quantization factor for Weight - signed 8bits

# For Weight
weight_list1 = np.around(weight_list1*Qw)
weight_list2 = np.around(weight_list2*Qw)
weight_list3 = np.around(weight_list3*Qw)
weight_list4 = np.around(weight_list4*Qw)

weight_list1 = weight_list1.astype(int)
weight_list2 = weight_list2.astype(int)
weight_list3 = weight_list3.astype(int)
weight_list4 = weight_list4.astype(int)

# For Sigmoid by Shift-add method
lower_bound = np.array([0.0, 1.065, 2.164, 2.977, 3.724, 4.442, 5.147, 5.846, 7.236])
constant = np.array([0.5, 0.6328125, 0.765625, 0.859375, 0.91796875, 0.953125, 0.97265625, 0.984375, 1.0])
sigmoid_max = np.array([1.0])

lower_bound = np.around(Qa*Qw+lower_bound) # signed 32 bits
constant = np.around(Qa*Qw+constant) # signed 32 bits
sigmoid_max = np.around(Qa*Qw+sigmoid_max)

lower_bound = np.around(Qa*Qw+lower_bound) # signed 32 bits
constant = np.around(Qa*Qw+constant) # signed 32 bits
sigmoid_max = np.around(Qa*Qw+sigmoid_max)

lower_bound = lower_bound.astype(int)
constant = constant.astype(int)
sigmoid_max = sigmoid_max.astype(int)

# For Test data in RTL
print("----- Check RTL data -----")
numOfTest = 10 # You can change how many sample you test
test_data = X_test[:numOfTest]
list_data = []
for data in test_data:
    for row in data:
        for col in row:
            list_data.append(col[0])
list_data = np.array(list_data)
list_data = np.around(list_data*Qa)
list_data = list_data.astype(int)
print(f'test_data.shape:{test_data.shape}')
print(f'list_data.shape:{list_data.shape}')
```

Fig 5

Weight도 같은 방식으로 maximum value를 구하고, 구한 값을 이용해 PTQ(Post Training Quantization)을 진행한다. Weight의 추출과 함께 Shift-add Sigmoid 함수의 lower boundary와 constant값을 구하기 위해 해당 값들도 quantization하여 출력한다. Verilog에서 Test를 위해 사용할 data의 경우 unsigned 8bits로 quantization하였다. 강의 시간에 주어진 코드를 이용하여 Test data와 weights는 모두 “<name>.hex”로 저장하였다. 이번 모델은 bias를 사용하지 않았다. 파라미터를 줄이기 위한 것이기도 했고, bias를 굳이 넣지 않아도 accuracy가 잘 나오기 때문이기도 했다.

3) 결과

```
Epoch 1/10
625/625 [=====] - 8s 13ms/step - loss: 0.2022 - accuracy: 0.6890 - val_loss: 0.3357 - val_accuracy: 0.4348
Epoch 2/10
625/625 [=====] - 8s 13ms/step - loss: 0.1716 - accuracy: 0.7494 - val_loss: 0.2153 - val_accuracy: 0.6598
Epoch 3/10
625/625 [=====] - 8s 13ms/step - loss: 0.1560 - accuracy: 0.7763 - val_loss: 0.2215 - val_accuracy: 0.6614
Epoch 4/10
625/625 [=====] - 8s 13ms/step - loss: 0.1399 - accuracy: 0.8029 - val_loss: 0.2436 - val_accuracy: 0.6346
Epoch 5/10
625/625 [=====] - 8s 13ms/step - loss: 0.1201 - accuracy: 0.8381 - val_loss: 0.2233 - val_accuracy: 0.6708
Epoch 6/10
625/625 [=====] - 8s 13ms/step - loss: 0.0949 - accuracy: 0.8806 - val_loss: 0.3233 - val_accuracy: 0.5542
Epoch 7/10
625/625 [=====] - 8s 13ms/step - loss: 0.0694 - accuracy: 0.9167 - val_loss: 0.3323 - val_accuracy: 0.5650
Epoch 8/10
625/625 [=====] - 8s 13ms/step - loss: 0.0476 - accuracy: 0.9459 - val_loss: 0.3148 - val_accuracy: 0.6004
Epoch 9/10
625/625 [=====] - 8s 13ms/step - loss: 0.0329 - accuracy: 0.9661 - val_loss: 0.2714 - val_accuracy: 0.6632
Epoch 10/10
625/625 [=====] - 8s 13ms/step - loss: 0.0250 - accuracy: 0.9743 - val_loss: 0.2674 - val_accuracy: 0.6720
```

Fig 6

Epoch 10번을 돌고 97.43%의 accuracy가 나오는데 모델을 반복적으로 돌려본 결과 최대 98.7%까지 accuracy가 보장된다. 레포트의 경우 Section 2. 의 Verilog에서는 현재 사용한 97.43%의 accuracy를 보이는 모델(Python level Worst case)을 기준으로 작성되었다.

```
Model: "sequential"
-----
Layer (type)                Output Shape              Param #
-----
conv2d (Conv2D)              (None, 56, 56, 16)       160
max_pooling2d (MaxPooling2D) (None, 28, 28, 16)       0
conv2d_1 (Conv2D)             (None, 26, 26, 32)       4640
max_pooling2d_1 (MaxPooling2D) (None, 13, 13, 32)       0
flatten (Flatten)            (None, 5408)              0
dense (Dense)                (None, 64)                346176
dense_1 (Dense)              (None, 1)                  65
-----
Total params: 351,041
Trainable params: 351,041
Non-trainable params: 0
-----
391/391 [=====] - 2s 4ms/step
```

Fig 7

출력한 model summary를 보면 사용한 전체 파라미터는 351041개인 것을 알 수 있다. 쉽게 찾아볼 수 있는 여러 예제 코드에 비해 훨씬 적은 파라미터를 사용하는 것을 알 수 있다. 보통 최소 백만 개 이상의 파라미터를 요구한다.

```
----- Maximum Value -----
Maximum Activations: 19.57419776916504
Maximum Weight: 0.8836168050765991
----- Check RTL data -----
test_data.shape:(20, 58, 58, 1)
list_data.shape:(1, 67280)
----- Check Factor -----
Qw: 143.72746112381896
Qa: 13.0273538158329
sigmoid_max:[[1872]]
lower_bound:
[[ 0 1994 4052 5574 6973 8317 9637 10946 13549]]
constant:
[[ 936 1185 1434 1609 1719 1785 1821 1843 1872]]
----- Check Shape -----
list_layer1: (1, 144)
lower_bound: (1, 9)
sigmoid_max: (1, 1)
----- Check Plot -----
```

Fig 8

추출한 quantization factor들이다. 32bits signed integer의 경우 1.0은 1872로 quantization되는 것을 알 수 있다. 이번 모델은 Maximum Activation value가 19.574였다.

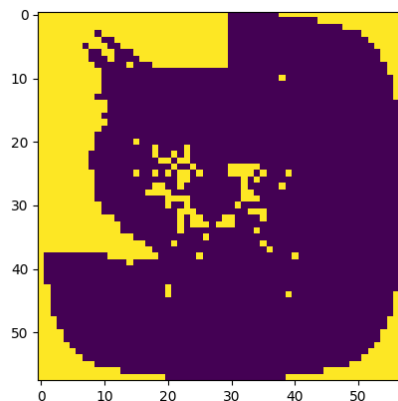


Fig 9

전 처리된 데이터 중에 무작위로 하나를 뽑아 출력하게 하였다. 이번에 출력된 이미지는 고양이 나 강아지가 분간이 안 되는 수준으로 처리되었다. 그럼에도 불구하고 최대 98.7%의 높은 accuracy를 보인 것은 사람은 인지할 수 없는 hidden features를 추출하는 Deep learning의 특징을 잘 보여주는 것이라고 생각한다.

2. Verilog

1) 설계 목표

일반적으로 Hardware를 설계할 때는 PPA(Performance, Power, Area)를 고려하여 가장 최선의 모델을 설계하게 된다. 우리는 RTL level에서의 설계만 진행하고 있기 때문에 Power와 Area는 결국 같은 목표를 갖는다. 일반적으로 low area는 lower power를 갖기 때문이다. 그래서 Performance(latency)와 area중 어떤 metrics를 목표로 할 것인지 정해야 했다. Latency를 줄이는 방법은 비교적 간단하다. 최대한 Python level에서 설계한 architecture와 동일한 모델을 설계하는 것이다. 모든 layer를 serial하게 이어 붙이고, 각 layer는 반복되는 구조를 parallel하게 나열한다. 그러나, 이 방식은 매우 큰 area를 요구한다. 이번 프로젝트의 경우 real-time(on-line) data를 이용하는 것이 아니기 때문에 low area를 목표로 하는 것이 더 적합하다고 판단하여 우리는 최대한 작은 area를 만족할 수 있는 모듈을 설계하기로 정하였다.

적은 area를 만족하기 위해서는 가장 적은 단위의 unit 모듈을 반복적으로 수행하도록 해야 한다. 이러한 memory base architecture의 경우 memory의 어느 위치에서 데이터를 가져와서 unit 모듈에 입력으로 넣고, 그 출력을 memory의 어느 위치에 넣어야 하는지를 결정하는 controller의 높은 complexity가 보장된다. 그러나, 최적의 경우 address를 계산하는 과정에서 conflict가 발생하지 않는다면 최소 1개의 unit 모듈만을 이용하여 구현할 수 있다. 우리는 1개의 unit 모듈이 될 PE를 구현하였고, 각 layer의 출력 값이 저장될 RAM, input data와 각 layer의 weight가 저장될 ROM을 구현하였다. 마지막으로, 각 RAM과 ROM에서 어떤 address의 data를 read하여 PE에서 연산 후 어떤 address에 PE에서 출력된 output을 write할지 정하는 controller를 구현하였다.

Memory base architecture의 경우 하나의 PE를 반복적으로 사용하기 때문에 area는 줄어들지만 latency가 매우 커지는 trade-off가 있다. 따라서 real-time data를 사용해야 하는 경우에는 적합하지 않다. 또한, 이번 프로젝트는 FPGA에 porting하지 않고, Modelsim만을 이용하여 waveform simulation을 하기 때문에 1회의 decision을 하는데, 매우 많은 시간이 소요된다는 단점이 존재했다. 결과적으로, 단계별로 address는 매우 잘 계산되었으며, 중간에 conflict도 발생하지 않았다. 이러한 controller는 FSM을 이용하여 구현되었으며, 각 layer별로 Input address calculation, data read, PE calculation, output address calculation, data write의 큰 flow에 따라서 FSM을 설계하였다. 단, 각 layer별로 중간에 삽입되는 state가 추가적으로 존재하며 이는 뒤에서 자세하게 설명할 것이다.

2) 모듈 분석

1. PE(Processing Element)

Processing element의 경우 강의 시간에 다룬 모듈들을 최대한 이용하기로 하였다.

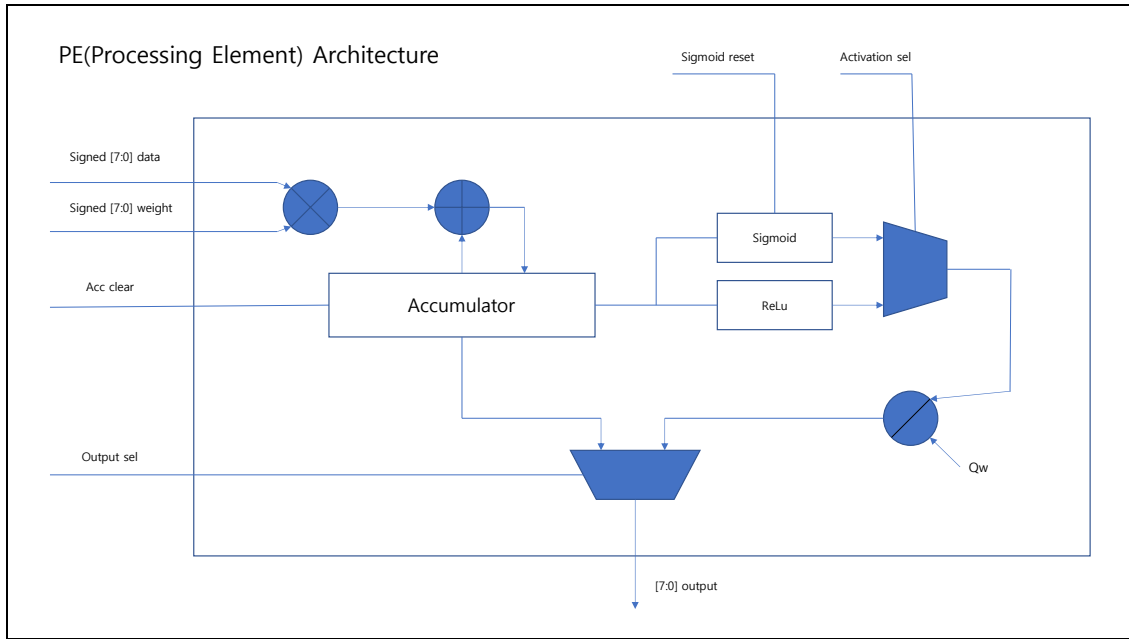


Fig 10

PE는 8bits signed data, 8bits signed weight를 입력으로 받아 multiplication을 진행한다. 그 후에 Accumulator에 누적해서 더해진다. Accumulator의 출력을 각각 sigmoid와 relu activation function의 입력으로 들어간다. 이때, activation selection신호에 의해 두 activation function중 하나의 출력 값이 2x1 MUX를 통해 출력된다. 출력된 값은 Quantize factor인 Qw로 나누어 지게 된다. 그 다음, output selection신호에 의해 accumulator의 출력과 divider의 출력 중 하나의 값을 출력한다.

먼저, 8bits system을 만족해야 하는데, 우리가 PTQ에서 학습한 바에 의하면 $x*w+b$ 는 32bits signed integer이다. 여기서 bias를 zero로 두면 $x*w$ 의 값은 8bits integer로 직접 사용할 수 없는 것은 당연하며, 모든 activation의 출력은 8bit로 정해져야 한다. 이때, accumulator의 출력 값은 단순히 $x*w$ 하나를 출력하는 것이 아니라 PE밖에서 몇번의 accumulation을 진행할 것인지 결정하여 누적된 값을 activation function을 거치게 하여 출력한다. 따라서 Qw로 나누어 unsigned 8bits를 output으로 출력하게 한다. 또한, accumulator의 출력을 output selection신호에 의해 그대로 출력하기도 하는데, 이는 다음과 같은 문제 때문에 고안되었다. 예를 들어, 2D convolution layer에서 입력 이미지가 16장이고, 출력되어야 하는 이미지는 32장이다. Kernel size가 3x3이라면, 하나의 kernel에 대해서 16장의 이미지의 같은 row, 같은 col을 가진 위치에 해당하는 모든 pixel이 2D convolution하여 누적된 값이 activation function을 지나게 된다. 즉, 마지막 이미지에 해당하는 값이 모두 kernel과 2D convolution을 진행하기 전까지는 activation function을 지난 값이 메모리에 저장되면 안 된다. 따라서 하나의 이미지에 대해서 32장의 커널들과 순차적으로 2D convolution을 한 pixel씩 진행하여 나온 출력 값이 메모리에 계속 저장되고, 16번째 이미지가 2D convolution된 후에는 activation function을 거친 값을 선택하여 메모리에 저장한다. PE의 코드는 다음과 같다.


```

1  module PE #(
2      parameter Qw = 123,
3      parameter sigmoid_max = 123)
4  (
5      input i_clk,
6      input i_rst,
7      input signed [7:0] i_data,
8      input signed [7:0] i_weight,
9      input i_op_activation, // 0 : ReLu, 1 : Sigmoid
10     input i_accumulator_clr,
11     input i_no_rect_quantize,
12     output reg [7:0] o_data,
13     output reg o_decision
14 );

```

Fig 11

PE의 입출력 포트, parameter는 top module에서 정해지며 현재는 임의의 값을 쓰고 있다. sigmoid_max는 python에서 구한 32bits signed integer의 1.0값을 의미한다. Architecture에서 소개되지 않은 출력이 하나 있는데, o_decision은 최종적인 decision을 위해 사용되며, o_data가 sigmoid_max / $Q_w / 2$ 이상이 되면 1을 출력하게 하여 강아지와 고양이를 구분하기 위한 신호로 사용된다.

```

15  ////////////////////////////////////////////////// Multiplier ///////////////////////////////////
16  reg signed [31:0] r_multi;
17  always @(*) begin
18      r_multi <= i_data * i_weight;
19  end
20  ////////////////////////////////////////////////// Accumulator ///////////////////////////////////
21  // Accumulator : 1 clk delay
22  wire signed [31:0] w_acc;
23  accumulator i_accumulator(
24      .i_clk(i_clk),
25      .i_accumulator_clr(i_accumulator_clr),
26      .i_data(r_multi),
27      .o_data(w_acc)
28  );
29  ////////////////////////////////////////////////// Activation ///////////////////////////////////
30  // ReLu : no delay, it's combinational logic
31  wire [31:0] w_relu;
32  ReLu i_relu(
33      .r_in(w_acc),
34      .r_out(w_relu)
35  );

```

Fig 12

Accumulator의 경우 sequential logic이기 때문에 1clk의 delay가 발생한다. ReLu의 경우 강의 시간에는 sequential logic으로 설계하였지만 delay를 줄이기 위해 combinational logic으로 교체하였다.

```

36 // Sigmoid : 4 clk delay, it's pipelined sequential logic
37 wire [31:0] w_sigmoid;
38 Sigmoid #(sigmoid_max) i_sigmoid(
39     .clk(i_clk),
40     .reset(i_rst),
41     .x(w_acc),
42     .o(w_sigmoid)
43 );
44 ////////////////////////////////////////////////// Select activation ///////////////////////////////////
45 reg [31:0] r_sel_act;
46 always @(*) begin
47     if(i_op_activation) begin
48         r_sel_act <= w_sigmoid;
49     end
50     else begin
51         r_sel_act <= w_relu;
52     end
53 end
54 ////////////////////////////////////////////////// Select output ///////////////////////////////////
55 always @(*) begin
56     if(i_no_rect_quantize) begin
57         o_data <= w_acc;
58     end
59     else begin
60         o_data <= r_sel_act / Qw;
61     end
62 end

```

Fig 13

Sigmoid activation은 강의 시간에 설계한 pipelined shift-add module을 그대로 사용하였다. 2개의 activation에 대한 2x1 MUX와 accumulator의 output과 activation function의 output중 하나를 선택하는 2x1 MUX가 설계되었다. 이때, activation의 출력은 Qw로 나누어 8bits unsigned integer가 출력되도록 한다.

```

63 ////////////////////////////////////////////////// decision ///////////////////////////////////
64 always @(*) begin
65     if(w_sigmoid >= sigmoid_max / 2) o_decision <= 1;
66     else o_decision <= 0;
67 end

```

Fig 14

마지막으로 o_decision은 sigmoid의 출력이 0.5에 해당하는 값 이상이 되면 1을 출력하도록 한다.

2. RAM

RAM과 ROM은 일반적으로 알려진 형태로 구현하였다.

```
1  module RAM #(
2      parameter addr_max = 123,
3      parameter addr_width = 123,
4      parameter data_width = 123
5  ) (
6      input i_clk,
7      input i_op, // , 0 : read mode, 1 : write mode
8      input [addr_width-1:0] i_addr,
9      input [data_width-1:0] i_data,
10     input i_mem_clr,
11     output reg [data_width-1:0] o_data
12 );
13 integer I;
14 reg [data_width-1:0] ram [0:addr_max-1];
15
16 always @(posedge i_clk) begin
17     if(i_mem_clr) begin
18         for(I=0;I<addr_max;I=I+1) begin
19             ram[I] = 0;
20         end
21     end
22     else begin
23         if(i_op) begin
24             ram[i_addr] <= i_data;
25         end
26         else begin
27             o_data <= ram[i_addr];
28         end
29     end
30 end
31 endmodule
```

Fig 15

여기서 addr_max, addr_width, data_width에 대한 parameter값은 임의로 설정하였으며 top module에서 결정한다. Address는 하나만 받도록 하였으며, enable신호도 하나만 받도록 하였다. 가장 간단한 형

태의 RAM구조이며, 조금 특이한 것은 reset신호에 대하여 모든 RAM의 값을 초기화할 때, for loop이 이용된 것을 알 수 있는데 for loop도 synthesis가 가능하기 때문에 위와 같이 설계하였다.

3. ROM

```
1  module ROM #(
2      parameter addr_max = 123,
3      parameter addr_width = 123,
4      parameter data_width = 123,
5      parameter INFILE = ""
6  ) (
7      input i_clk,
8      input [addr_width-1:0] i_addr,
9      output reg [data_width-1:0] o_data
10 );
11 reg [data_width-1:0] rom [0:addr_max-1];
12
13 initial begin
14     $readmemh(INFILE,rom);
15 end
16
17 always @(posedge i_clk) begin
18     o_data <= rom[i_addr];
19 end
20
21 endmodule
```

Fig 16

ROM도 RAM과 같은 가장 간단한 구조로 설계하였으며, system task인 \$readmemh를 이용하여 초기화하였다. 마찬가지로, parameter INFILE은 top module에서 결정한다.

4. Controller

Memory base architecture의 핵심이 되는 controller의 경우 complexity가 매우 높다. 코드를 따라가며 순서대로 개괄적으로 설명한 후 address control의 핵심이 되는 알고리즘을 집중적으로 설명하도록 할 것이다.

```
15  module Controller #(
16      parameter Qw = 142,
17      parameter sigmoid_max = 1564,
18      parameter TESTNUMBER = 1,
19      parameter ROM0_ADDR_MAX = 3364,
20      parameter ROM0_ADDR_WIDTH = 12,
21      parameter INFILE0 = "RTL_test_data.hex",
22      parameter INFILE1 = "weights_layer1.hex",
23      parameter INFILE2 = "weights_layer2.hex",
24      parameter INFILE3 = "weights_layer3.hex",
25      parameter INFILE4 = "weights_layer4.hex"
26  ) (
27      input i_clk,
28      input i_rst,
29      input i_en,
30      output reg [19:0] o_array, // 0 : cat, 1 : dog
31      output reg done // 1 : done
32  );
```

Fig 17

Controller의 parameter는 현재 임의로 설정하였으며, top module에서 parameter들의 value를 결정한다. 여기서 ROM0는 test data를 내장한 ROM이다. Test data의 size가 58x58이기 때문에 하나의 이미지는 총 3364개의 pixel을 갖고 하나의 pixel당 하나의 address를 갖기 때문에 ROM0_ADDR_MAX는 3364로 설정되었다. 입력 시그널인 i_en은 FSM을 idle상태에서 동작 상태로 transition하기 위한 signal로 사용된다. 출력 시그널 중 done은 한 번의 decision이 끝날 때마다 high가 된다. 출력 중 o_array는 20bit로 되어 있는데, 이는 20개의 test sample을 가지고, 수행했기 때문에 결과를 편하게 보기 위해서 array에 left shift시키며 저장했기 때문이다.

```

33  //////////// FOR TEST ////////////
34  reg [4:0] numberOftest;
35  //////////// PARAMETER ////////////
36  localparam RAM1_ADDR_MAX = 12544;
37  localparam RAM1_ADDR_WIDTH = 14;
38  localparam RAM2_ADDR_MAX = 5408;
39  localparam RAM2_ADDR_WIDTH = 13;
40  localparam RAM3_ADDR_MAX = 64;
41  localparam RAM3_ADDR_WIDTH = 6;
42  localparam ROM1_ADDR_MAX = 144;
43  localparam ROM1_ADDR_WIDTH = 8;
44  localparam ROM2_ADDR_MAX = 288;
45  localparam ROM2_ADDR_WIDTH = 9;
46  localparam ROM3_ADDR_MAX = 346112;
47  localparam ROM3_ADDR_WIDTH = 19;
48  localparam ROM4_ADDR_MAX = 64;
49  localparam ROM4_ADDR_WIDTH = 6;
50  localparam DATA_WIDTH = 8;

```

Fig 18

numberOftest 레지스터는 몇 번째 입력 이미지인지를 나타내기 위한 counter로 동작하며, 입력 이미지의 address를 결정하기 위해 사용된다. 35번라인부터 50번라인까지 설정된 모든 파라미터는 이후 수정없이 모두 사용될 수 있기 때문에 localparam으로 두었다. 먼저, layer1에서는 58x58이미지가 입력으로 들어와 16개의 28x28이미지를 출력한다. 여기서 28x28인 이유는 동시에 2D convolution과 max pooling을 진행하기 때문이다. 따라서 RAM은 12544개의 address line을 가져야 한다. 이에 해당하는 address width는 14이다. 2번째 layer의 경우 출력으로 32장의 13x13이미지가 결정되기 때문에 5408개의 address line이 필요하고, 이때, address width는 13이다. 3번째 layer는 FC로 64개의 neuron에 대응된다. 따라서 64개의 address line이 필요하고, 이때, 필요한 address width는 6이다. ROM의 경우 ROM1은 layer1에 사용되는 weight들을 담고 있다. Weight의 경우 하나의 kernel당 9개의 weight가 필요하며, kernel은 총 16개로 weight를 담고 있는 룬은 총 144개의 weight를 내장해야 한다. 따라서 이때, 필요한 address line은 144이며, address width는 8이 되어야 한다. 2번째, layer는 32개의 kernel을 사용하기 때문에 address line이 288개가 필요하며, address width는 9가 된다. 3번째 layer는 fully connected layer로 입력으로 들어오는 neuron의 개수는 32x13x13으로 5408개이고, 출력으로 대응되는 neuron은 64개이다. 따라서 필요한 모든 address line은 5408x64로 346112개가 된다. ROM3에서 매우 많은 address line이 요구되는 것을 알 수 있으며, 이 때문에 우리는 강의에서 Fully connected layer를 verilogA를 통해 설계하는 법을 배운다고 생각한다. 이때, 필요한 address width는 19이다. 마지막으로 output layer의 경우 64개의 뉴런을 입력으로 받아 1개의 뉴런으로 출력되므로 필요한

address line은 64이고, 이때 필요한 address width는 6이다. 우리는 INT8로 quantization하여 설계하고 있기 때문에 DATA_WIDTH는 8이 된다.

```
52 // PE
53 reg [DATA_WIDTH-1:0] i_pe_data;
54 reg signed [DATA_WIDTH-1:0] i_pe_weight;
55 reg i_op_activation;
56 reg i_accumulator_clr;
57 reg i_no_rect_quantize;
58 wire [DATA_WIDTH-1:0] o_pe_data;
59 wire o_decision;
```

Fig 19

PE에 사용되는 입출력 포트이다.

```
60 // RAM
61 reg i_mem_clr;
62 reg i_ram1_op;
63 reg [RAM1_ADDR_WIDTH-1:0] i_ram1_addr;
64 reg [DATA_WIDTH-1:0] i_ram1_data;
65 wire [DATA_WIDTH-1:0] o_ram1_data;
66 reg i_ram2_op;
67 reg [RAM2_ADDR_WIDTH-1:0] i_ram2_addr;
68 reg [DATA_WIDTH-1:0] i_ram2_data;
69 wire [DATA_WIDTH-1:0] o_ram2_data;
70 reg i_ram3_op;
71 reg [RAM3_ADDR_WIDTH-1:0] i_ram3_addr;
72 reg [DATA_WIDTH-1:0] i_ram3_data;
73 wire [DATA_WIDTH-1:0] o_ram3_data;
```

Fig 20

RAM에 사용되는 입출력 포트이다. 총 3개의 RAM이 instantiation되는 것을 알 수 있다.

```

74 // ROM
75 reg [ROM0_ADDR_WIDTH-1:0] i_rom0_addr;
76 wire [DATA_WIDTH-1:0] o_rom0_data;
77 reg [ROM1_ADDR_WIDTH-1:0] i_rom1_addr;
78 wire [DATA_WIDTH-1:0] o_rom1_data;
79 reg [ROM2_ADDR_WIDTH-1:0] i_rom2_addr;
80 wire [DATA_WIDTH-1:0] o_rom2_data;
81 reg [ROM3_ADDR_WIDTH-1:0] i_rom3_addr;
82 wire [DATA_WIDTH-1:0] o_rom3_data;
83 reg [ROM4_ADDR_WIDTH-1:0] i_rom4_addr;
84 wire [DATA_WIDTH-1:0] o_rom4_data;

```

Fig 21

ROM에 사용되는 입출력 포트이다. 총 4개의 ROM이 instantiation되는 것을 알 수 있다.

```

85 // CONTROL FLAG
86 reg [5:0] row;
87 reg [5:0] col;
88 reg [1:0] cnt_row;
89 reg [1:0] cnt_col;
90 reg [RAM2_ADDR_WIDTH-1:0] cnt_accumulator;
91 reg [2:0] cnt_pe;
92 reg [ROM0_ADDR_WIDTH-1:0] ram_write_addr;
93 reg [3:0] cnt_image;
94 reg [RAM3_ADDR_WIDTH-1:0] cnt_ram3;
95 reg [4:0] addr_start;
96 reg [3:0] cnt_weight;
97 reg [ROM3_ADDR_WIDTH-1:0] cnt_rom3;

```

Fig 22

FSM내에서 address를 결정하고, state를 transition하기 위해 사용되는 counter들이다. 각각의 역할에 대한 설명은 각 state에서 설명하도록 한다.


```

98  //////////// INTERNAL CONNECTION ////////////
99  PE #(Qw, sigmoid_max) i_PE(
100      .i_clk(i_clk),
101      .i_rst(i_rst),
102      .i_data(i_pe_data),
103      .i_weight(i_pe_weight),
104      .i_op_activation(i_op_activation),
105      .i_accumulator_clr(i_accumulator_clr),
106      .i_no_rect_quantize(i_no_rect_quantize),
107      .o_data(o_pe_data),
108      .o_decision(o_decision)
109  );

```

Fig 23

하나의 PE를 instantiation하였다. 우리는 단 1개의 PE를 이용하여 CNN을 구현한다.

```

110  RAM #(RAM1_ADDR_MAX, RAM1_ADDR_WIDTH, DATA_WIDTH) i_RAM1(
111      .i_clk(i_clk),
112      .i_op(i_ram1_op),
113      .i_addr(i_ram1_addr),
114      .i_data(i_ram1_data),
115      .i_mem_clr(i_mem_clr),
116      .o_data(o_ram1_data)
117  );
118  RAM #(RAM2_ADDR_MAX, RAM2_ADDR_WIDTH, DATA_WIDTH) i_RAM2(
119      .i_clk(i_clk),
120      .i_op(i_ram2_op),
121      .i_addr(i_ram2_addr),
122      .i_data(i_ram2_data),
123      .i_mem_clr(i_mem_clr),
124      .o_data(o_ram2_data)
125  );
126  RAM #(RAM3_ADDR_MAX, RAM3_ADDR_WIDTH, DATA_WIDTH) i_RAM3(
127      .i_clk(i_clk),
128      .i_op(i_ram3_op),
129      .i_addr(i_ram3_addr),
130      .i_data(i_ram3_data),
131      .i_mem_clr(i_mem_clr),
132      .o_data(o_ram3_data)
133  );

```

Fig 24

3개의 RAM이 instantiation되었다. 각 RAM은 hidden layer의 출력 값을 저장한다.

```
134 ROM #(ROM0_ADDR_MAX, ROM0_ADDR_WIDTH, DATA_WIDTH, INFILE0) i_ROM0(  
135     .i_clk(i_clk),  
136     .i_addr(i_rom0_addr),  
137     .o_data(o_rom0_data)  
138 ); // TEST DATA ROM  
139 ROM #(ROM1_ADDR_MAX, ROM1_ADDR_WIDTH, DATA_WIDTH, INFILE1) i_ROM1(  
140     .i_clk(i_clk),  
141     .i_addr(i_rom1_addr),  
142     .o_data(o_rom1_data)  
143 ); // WEIGHTS OF LAYER1  
144 ROM #(ROM2_ADDR_MAX, ROM2_ADDR_WIDTH, DATA_WIDTH, INFILE2) i_ROM2(  
145     .i_clk(i_clk),  
146     .i_addr(i_rom2_addr),  
147     .o_data(o_rom2_data)  
148 ); // WEIGHTS OF LAYER2  
149 ROM #(ROM3_ADDR_MAX, ROM3_ADDR_WIDTH, DATA_WIDTH, INFILE3) i_ROM3(  
150     .i_clk(i_clk),  
151     .i_addr(i_rom3_addr),  
152     .o_data(o_rom3_data)  
153 ); // WEIGHTS OF LAYER3  
154 ROM #(ROM4_ADDR_MAX, ROM4_ADDR_WIDTH, DATA_WIDTH, INFILE4) i_ROM4(  
155     .i_clk(i_clk),  
156     .i_addr(i_rom4_addr),  
157     .o_data(o_rom4_data)  
158 ); // WEIGHTS OF LAYER4
```

Fig 25

4개의 ROM이 instantiation되었다. ROM0은 test data를 ROM1은 layer1의 weight를 ROM2는 layer2의 weight를 ROM3은 layer3의 weight를 ROM4는 output layer의 weight를 담고 있다.

```

159  //////////// STATE ////////////
160  reg [4:0] r_ps;
161  localparam ST_IDLE = 0;
162  localparam ST_LAYER1_INPUT_ADDR = 1;
163  localparam ST_LAYER1_READ = 2;
164  localparam ST_LAYER1_CALCULATE = 3;
165  localparam ST_LAYER1_HOLD = 4;
166  localparam ST_LAYER1_OUTPUT_ADDR = 5;
167  localparam ST_LAYER1_WRITE = 6;
168  localparam ST_LAYER2_INPUT_ADDR = 7;
169  localparam ST_LAYER2_READ = 8;
170  localparam ST_LAYER2_CALCULATE = 9;
171  localparam ST_LAYER2_HOLD = 10;
172  localparam ST_LAYER2_OUTPUT_ADDR = 11;
173  localparam ST_LAYER2_READ2 = 12;
174  localparam ST_LAYER2_HOLD2 = 13;
175  localparam ST_LAYER2_WRITE = 14;
176  localparam ST_LAYER3_INPUT_ADDR = 15;
177  localparam ST_LAYER3_READ = 16;
178  localparam ST_LAYER3_CALCULATE = 17;
179  localparam ST_LAYER3_HOLD = 18;
180  localparam ST_LAYER3_OUTPUT_ADDR = 19;
181  localparam ST_LAYER3_WRITE = 20;
182  localparam ST_LAYER4_INPUT_ADDR = 21;
183  localparam ST_LAYER4_READ = 22;
184  localparam ST_LAYER4_CALCULATE = 23;
185  localparam ST_LAYER4_HOLD = 24;
186  localparam ST_LAYER4_DECISION = 25;
187  localparam ST_DONE = 26;

```

Fig 26

총 27개의 state가 선언되었다. 다음은 모든 state에 대한 정의이다.

ST_IDLE: 대기상태

ST_LAYER1_INPUT_ADDR: ROM0와 ROM1에서 데이터를 가져오기 위해 address를 계산하는 상태

ST_LAYER1_READ: address계산 후 data read에 필요한 1clk을 기다리기 위한 상태

ST_LAYER1_CALCULATE: ROM0와 ROM1에서 가져온 데이터를 PE에 넣어주는 상태

ST_LAYER1_HOLD: PE에서 출력이 나올 때까지 기다리기 위한 상태

ST_LAYER1_OUTPUT_ADDR: PE에서 나온 출력을 RAM1에 저장하기 위해 RAM1의 address를 계산하는 상태

ST_LAYER1_WRITE: RAM1에 data가 write되기 위해 1clk을 기다리기 위한 상태

ST_LAYER2_INPUT_ADDR: RAM1과 ROM2에서 데이터를 가져오기 위해 address를 계산하는 상태

ST_LAYER2_READ: data read에 필요한 1clk을 기다리기 위한 상태

ST_LAYER2_CALCULATE: PE에 RAM1과 ROM2에서 가져온 데이터를 넣어 주는 상태

ST_LAYER2_HOLD: PE에서 계산이 끝날 때까지 기다리기 위한 상태

ST_LAYER2_OUTPUT_ADDR: PE의 출력을 RAM2에 쓰기 위해 address를 계산하는 상태

ST_LAYER2_READ2: layer2는 마지막 사진이 되기 전까지는 data를 누적해서 더해야 하기 때문에 앞의 state에서 구한 address의 data를 read하기 위해 1clk을 기다리는 상태

ST_LAYER2_HOLD2: RAM2에 앞서 PE의 출력과 RAM2에서 read한 데이터를 넣어주는 상태

ST_LAYER2_WRITE: RAM2에 data가 write되기 위해 1clk을 기다리는 상태

ST_LAYER3_INPUT_ADDR: RAM2와 ROM3에서 데이터를 가져오기 위해 address를 계산하는 상태

ST_LAYER3_READ: data를 read하기 위해 1clk을 기다리는 상태

ST_LAYER3_CALCULATE: 가져온 data를 PE에 넣어주는 상태

ST_LAYER3_HOLD: PE의 연산이 끝날 때까지 기다려주기 위한 상태

ST_LAYER3_OUTPUT_ADDR: RAM3에 data를 쓰기 위해 address를 계산하는 상태

ST_LAYER3_WRITE: data가 write되기 위해 1clk을 기다리는 상태

ST_LAYER4_INPUT_ADDR: RAM3와 ROM4에서 data를 가져오기 address를 계산하는 상태

ST_LAYER4_READ: data를 read하기 위해 1clk을 기다리는 상태

ST_LAYER4_CALCULATE: 가져온 data를 PE에 넣어주는 상태

ST_LAYER4_HOLD: PE에서 출력이 결정될 때까지 기다리는 상태

ST_LAYER4_DECISION: PE의 출력을 바탕으로 최종 decision을 하는 상태

ST_DONE: 1회의 decision이 끝난 후 모든 초기화를 진행하고, 다음 이미지를 가져오기 위한 카운

터를 동작시키는 상태

이제 모든 state에 대한 정의가 끝났기 때문에 각 state별로 어떤 동작을 수행하는지 알아보도록 한다. Reset은 생략하고 IDLE state부터 설명한다.

```
242         case (r_ps)
243             ST_IDLE :
244                 begin
245                     if(i_en) begin
246                         r_ps <= ST_LAYER1_INPUT_ADDR;
247                         i_accumulator_clr <= 1;
248                         i_mem_clr <= 1;
249                     end
250                 end
```

Fig 27

ST_IDLE에서는 i_en이 들어오면 ST_LAYER1_INPUT_ADDR로 state를 transition시키고, 동시에 accumulator와 RAM을 초기화 시킨다. i_mem_clr는 모든 RAM이 공통으로 사용한다.

```
251         ST_LAYER1_INPUT_ADDR :
252             begin
253                 r_ps <= ST_LAYER1_READ;
254                 i_accumulator_clr <= 0;
255                 i_mem_clr <= 0;
256                 // ROM1
257                 i_rom1_addr <= 9 * addr_start + cnt_weight;
258                 if(cnt_weight == 8) begin
259                     cnt_weight <= 0;
260                 end
261                 else begin
262                     cnt_weight <= cnt_weight + 1;
263                 end
```

Fig 28

ST_LAYER1_INPUT_ADDR에서는 state를 ST_LAYER1_READ로 transition시키고, 동시에 accumulator와 RAM의 초기화를 종료한다. ROM1에는 144개의 weight가 순서대로 저장되어 있다. 이때, cnt_weight는 ST_LAYER1_INPUT_ADDR이 될 때마다 하나씩 증가하고, 8이 되면, 다시 0으로 돌아간다. addr_start는 몇 번째 kernel인지를 나타내는 counter이다. addr_start는 뒤의 state에서도 사용되기 때문에 동작을 뒤에서 기술한다. addr_start와 cnt_weight를 이용하여 정확하게 ROM1의 read address를 계산하게 된다.

```

264         // ROM0
265         i_rom0_addr <= (ROM0_ADDR_MAX / TESTNUMBER) * numberOftest + 58 * row + col;
266         if(cnt_col == 2) begin
267             cnt_col <= 0;
268             if(cnt_row == 2) begin
269                 cnt_row <= 0;
270                 if(col == 56) begin
271                     row <= row;
272                     col <= 0;
273                 end
274                 else begin
275                     row <= row - 2;
276                     col <= col;
277                 end
278             end
279             else begin
280                 cnt_row <= cnt_row + 1;
281                 row <= row + 1;
282                 col <= col - 2;
283             end
284         end
285         else begin
286             cnt_col <= cnt_col + 1;
287             col <= col + 1;
288         end
289     end

```

Fig 29

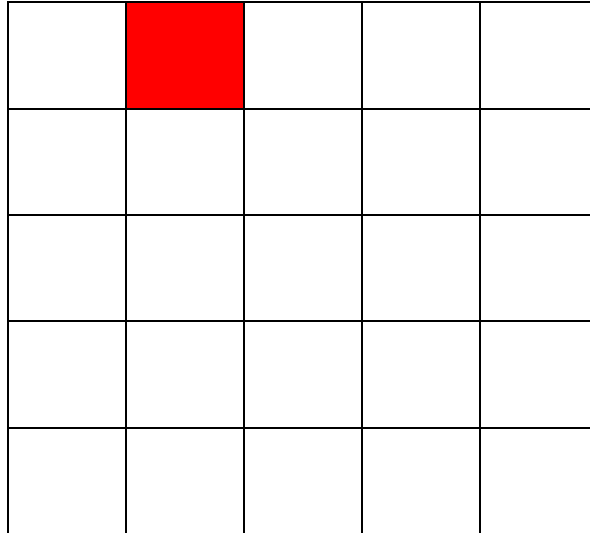
또한, 동시에 ROM0의 read address를 결정한다. 이것이 가장 핵심적인 알고리즘이기 때문에 다음을 통해 이해하도록 한다.

Assume. 다음과 같은 5x5입력에 3x3 kernel을 2D convolution한다고 생각하자.

우리의 architecture는 하나의 pixel씩 연산하기 때문에 가장 처음 연산 되는 것은 (1,1)일 것이다.

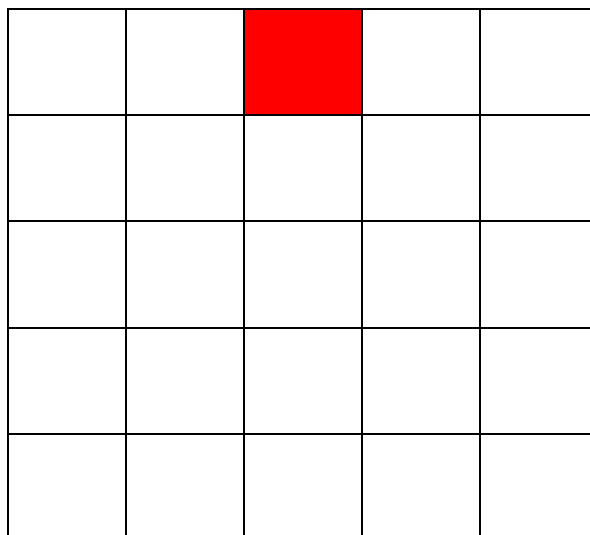
Step1. Col + 1

그 다음 pixel은 (1,2)가 될 것이다.



Step2. Col+1

그 다음 pixel은 (1,3)이 될 것이다.



Step3. Col-2, Row+1

여기서 우리가 알 수 있는 것은 col이 2번 증가한 후에는 다시 2만큼 감소하고 row를 1증가시킨다는 것이다.

Step4. Col+1

Step5. Col+1

Step6. Col-2, Row+1

위의 3개의 Step은 처음과 동일하다.

Step7. Col+1

Step8. Col+1

위의 2개의 Step도 마찬가지다.

Step9. Row-2

여기서 주의할 것은 (1,2)가 아닌 (1,3)으로 옮겼다는 것이다. 이는 2D convolution의 stride를 2로 함으로써 2D convolution stride 1 + maxpooling stride2를 한 것과 등가의 연산이 수행된다.

아직 봐야할 것이 더 있다. 현재 상태에서 8번의 동일한 Step을 적용하면 다음과 같이 될 것이다.

이때, Col이 가장 마지막 열에 도착한 경우에는 Col을 가장 작은 값으로 초기화 시킨다.

결과는 다음과 같다. 지금까지 이해한 바를 바탕으로 설계를 해야 한다. Col과 Row는 각각 counter임을 알 수 있다. 이때, Col이 몇 칸을 움직였는지, Row가 몇 칸을 움직였는지에 따라서 다음 움직임이 결정된다. 따라서 Col과 Row가 몇 칸을 움직였는지를 세야 함으로 각각의 index가 되는 col_cnt, row_cnt를 통해 컨트롤 해야 한다.

정리하면 cnt_col이 2가 되면 cnt_col을 0으로 초기화하고, 이때, cnt_row가 2라면 cnt_row를 0으로 초기화한다. 그렇지 않으면 cnt_row를 1증가시키고 동시에 col은 2만큼 감소, row는 1만큼 증가한다. 만약, cnt_col, cnt_row가 2일 때, col이 56에 도달한 경우 col을 0으로 초기화하고, 그렇지 않은 경우 row를 2만큼 감소시킨다. 265번 라인을 다시 보면 다음과 같다.

```
265 i_rom0_addr <= (ROM0_ADDR_MAX / TESTNUMBER) * numberOftest + 58 * row + col;
```

Fig 30

여기서 (ROM0_ADDR_MAX / TESTNUMBER)는 3364로 0번째 이미지에서 1번째 이미지가 되면 그 시작주소는 3364가 되기 때문에 n번째 이미지의 시작 위치를 나타내기 위해 사용된다. numberOftest는 FSM이 모두 수행되어 ST_LAYER4_DONE이 되었을 때, 1씩 증가한다.

```
290 ST_LAYER1_READ :
291 begin
292     r_ps <= ST_LAYER1_CALCULATE;
293 end
```

Fig 31

계산한 address에 대한 출력이 나오는 시간 1clk을 기다리기 위해 사용되고, ST_LAYER1_CALCULATE으로 transition한다.

```

294         ST_LAYER1_CALCULATE :
295         begin
296             r_ps <= ST_LAYER1_HOLD;
297             i_pe_data <= o_rom0_data;
298             i_pe_weight <= o_rom1_data;
299             i_op_activation <= 0;
300         end

```

Fig 32

ST_LAYER1_HOLD로 transition하는 동시에 ROM0, ROM1에서 출력되는 값을 PE에 입력으로 넣는다. 이때, activation을 relu로 선택한다.

```

301         ST_LAYER1_HOLD :
302         begin
303             i_pe_data <= 0;
304             i_pe_weight <= 0;
305             if(cnt_pe == 1) begin
306                 r_ps <= ST_LAYER1_OUTPUT_ADDR;
307                 cnt_pe <= 0;
308             end
309             else begin
310                 cnt_pe <= cnt_pe + 1;
311             end
312         end

```

Fig 33

Accumulator에 들어가는 값을 0으로 바꿔 accumulator가 계속 증가하는 것을 방지한다. Relu는 accumulator에서 1clk만 delay가 걸리기 때문에 cnt_pe가 1이 되면 cnt_pe를 0으로 하고, ST_LAYER1_OUTPUT_ADDR로 transition한다.

```

313      ST_LAYER1_OUTPUT_ADDR :
314      begin
315          if(cnt_accumulator == 8) begin
316              r_ps <= ST_LAYER1_WRITE;
317              i_ram1_op <= 1;
318              i_ram1_data <= o_pe_data;
319              i_ram1_addr <= 784 * addr_start + ram_write_addr;
320              i_accumulator_clr <= 1;
321              cnt_accumulator <= 0;
322          end
323      else begin
324          r_ps <= ST_LAYER1_INPUT_ADDR;
325          cnt_accumulator <= cnt_accumulator + 1;
326      end
327  end

```

Fig 34

RAM1에 값을 쓰기 위해서는 PE가 9번 동작해야 한다. cnt_accumulator를 통해 몇 번 PE가 동작했는지를 세며, 8이 되지 않았을 경우에는 다시 ST_LAYER1_INPUT_ADDR로 돌아간다. cnt_pe가 8이 되어 9번 모두 수행되면, RAM1을 write모드로 하고, RAM1에 PE의 출력을 넣는다. RAM1의 address는 $784(28 \times 28) \times \text{addr_start} + \text{ram_write_addr}$ 로 한다. 하나의 이미지와 하나의 kernel이 2D convolution stride2를 진행하면 784개의 pixel에 해당하는 값이 출력되는데, addr_start는 몇 번째 kernel인지를 나타낸다. ram_write_addr는 783까지 동작하는 카운터이며, 후에 설명한다. PE에서 출력을 꺼냈기 때문에 동시에 accumulator를 초기화 시키고, cnt_accumulator를 0으로 한다. ST_LAYER1_WRITE로 transition한다.

```

328      ST_LAYER1_WRITE :
329      begin
330          i_ram1_op <= 0;
331          i_accumulator_clr <= 0;
332          if(ram_write_addr == 783) begin
333              row <= 0;
334              col <= 0;
335              cnt_row <= 0;
336              cnt_col <= 0;
337              cnt_pe <= 0;
338              cnt_accumulator <= 0;
339              cnt_weight <= 0;
340              ram_write_addr <= 0;
341
342              if(addr_start == 15) begin
343                  r_ps <= ST_LAYER2_INPUT_ADDR;
344                  addr_start <= 0;
345                  i_rom1_addr <= 0;
346                  i_rom0_addr <= 0;
347                  i_ram1_addr <= 0;
348              end
349              else begin
350                  r_ps <= ST_LAYER1_INPUT_ADDR;
351                  addr_start <= addr_start + 1;
352              end
353          end
354          else begin
355              r_ps <= ST_LAYER1_INPUT_ADDR;
356              ram_write_addr <= ram_write_addr + 1;
357          end
358      end

```

Fig 35

RAM1을 read mode로 돌려놓고, accumulator의 초기화를 종료한다. ram_write_addr 카운터를 동작 시켜 783이 되면, 하나의 이미지에 대한 연산이 끝난 것이기 때문에, row, col, cnt_row, cnt_col, cnt_pe, cnt_accumulator, cnt_weight, ram_write_addr를 모두 초기화 시킨다. 동시에 다음 kernel과 연산해야 하므로 addr_start를 동작 시킨다. 이때, addr_start가 15가 되어 16장의 kernel이 모두 연산이 되면,

ST_LAYER2_INPUT_ADDR로 transition하며, addr_start, i_rom1_addr, i_rom0_addr, i_ram1_addr을 모두 초기화 한다. addr_start가 15에 도달하지 못한 경우 다시 ST_LAYER1_INPUT_ADDR로 돌아가면서 addr_start를 1증가시킨다. 만약 ram_write_addr이 783이 되지 못해서 아직 하나의 kernel에 대한 연산이 모두 수행되지 않았다면 ST_LAYER1_INPUT_ADDR로 돌아가면서 ram_write_addr을 1증가시킨다.

지금 까지가 layer1에 대한 모든 state의 동작이었다. Layer2도 2D convolution이기 때문에 layer1와 매우 유사하지만 다른 점은 layer2의 경우 입력 이미지가 1장이 아니라 16장이라는 것이다. 이 차이를 주기 위해 input address를 계산할 때, 몇 번째 사진인지를 나타내 주는 counter가 하나 추가되면 된다.

```
359      ST_LAYER2_INPUT_ADDR :  
360      begin  
361          // ROM  
362          i_rom2_addr <= 9 * addr_start + cnt_weight;  
363          if(cnt_weight == 8) begin  
364              cnt_weight <= 0;  
365          end  
366          else begin  
367              cnt_weight <= cnt_weight + 1;  
368          end
```

Fig 36

롬의 경우 바뀐 것은 ROM2로 바뀐다는 것뿐이므로 ST_LAYER1_INPUT_ADDR과 동작은 동일하다.

```

369          // RAM
370          r_ps <= ST_LAYER2_READ;
371          i_ram1_op <= 0;
372          i_ram1_addr <= 169 * cnt_image + 28 * row + col;
373          if(cnt_col == 2) begin
374              cnt_col <= 0;
375              if(cnt_row == 2) begin
376                  cnt_row <= 0;
377                  if(col == 26) begin
378                      row <= row;
379                      col <= 0;
380                  end
381                  else begin
382                      row <= row - 2;
383                      col <= col;
384                  end
385              end
386              else begin
387                  cnt_row <= cnt_row + 1;
388                  row <= row + 1;
389                  col <= col - 2;
390              end
391          end
392          else begin
393              cnt_col <= cnt_col + 1;
394              col <= col + 1;
395          end
396          end

```

Fig 37

다른 동작은 모두 동일하며 $169(13 \times 13) \times \text{cnt_image}$ 가 추가되어 몇 번째 이미지인지를 나타내 주기 위한 counter가 하나 추가되었다.

```

397          ST_LAYER2_READ :
398          begin
399              r_ps <= ST_LAYER2_CALCULATE;
400          end

```

Fig 38

ST_LAYER1_READ와 동일하다.

```

401      ST_LAYER2_CALCULATE :
402      begin
403          r_ps <= ST_LAYER2_HOLD;
404          i_pe_data <= o_ram1_data;
405          i_pe_weight <= o_rom2_data;
406          i_op_activation <= 0;
407      end

```

Fig 39

ST_LAYER1_CALCULATE와 동일하다.

```

408      ST_LAYER2_HOLD :
409      begin
410          i_pe_data <= 0;
411          i_pe_weight <= 0;
412          if(cnt_pe == 1) begin
413              r_ps <= ST_LAYER2_OUTPUT_ADDR;
414              cnt_pe <= 0;
415          end
416          else begin
417              cnt_pe <= cnt_pe + 1;
418          end
419      end

```

Fig 40

ST_LAYER1_HOLD와 동일하다.

```

420      ST_LAYER2_OUTPUT_ADDR :
421      begin
422          if(cnt_accumulator == 8) begin
423              r_ps <= ST_LAYER2_READ2;
424              cnt_accumulator <= 0;
425              i_ram2_addr <= 169 * addr_start + ram_write_addr;
426          end
427          else begin
428              r_ps <= ST_LAYER2_INPUT_ADDR;
429              cnt_accumulator <= cnt_accumulator + 1;
430          end
431      end

```

Fig 41

ST_LAYER1_OUTPUT_ADDR과 동일하다.


```

432      ST_LAYER2_READ2 :
433      begin
434          r_ps <= ST_LAYER2_HOLD2;
435          if(cnt_image == 15) begin
436              i_no_rect_quantize <= 0;
437          end
438          else begin
439              i_no_rect_quantize <= 1;
440          end
441      end

```

Fig 42

마지막 장이 입력으로 들어온 것이 아니라면 activation을 하지 않고, 누적해서 더해야 하기 때문에 read에 필요한 1clk을 기다림과 동시에 i_no_rect_quantize를 1로 하여 PE에서 나오는 출력이 accumulator의 값이 되도록 한다.

```

442      ST_LAYER2_HOLD2 :
443      begin
444          i_ram2_op <= 1;
445          i_ram2_data <= o_ram2_data + o_pe_data;
446          i_accumulator_clr <= 1;
447          r_ps <= ST_LAYER2_WRITE;
448      end

```

Fig 43

Address는 이미 계산되었기 때문에 RAM2를 write mode로 바꾸고, RAM2에서 출력되는 값에 PE의 값을 더해 입력으로 넣는다. 동시에 accumulator를 초기화 시키고 ST_LAYER2_WRITE로 transition 한다.

```

449      ST_LAYER2_WRITE :
450      begin
451          i_ram2_op <= 0;
452          i_accumulator_clr <= 0;
453          if(ram_write_addr == 168) begin
454              row <= 0;
455              col <= 0;
456              cnt_row <= 0;
457              cnt_col <= 0;
458              cnt_pe <= 0;
459              cnt_accumulator <= 0;
460              cnt_weight <= 0;
461              ram_write_addr <= 0;
462              if(addr_start == 31) begin
463                  addr_start <= 0;
464                  if (cnt_image == 15) begin
465                      r_ps <= ST_LAYER3_INPUT_ADDR;
466                      i_rom2_addr <= 0;
467                      i_ram1_addr <= 0;
468                      i_ram2_addr <= 0;
469                      cnt_image <= 0;
470                  end
471                  else begin
472                      r_ps <= ST_LAYER2_INPUT_ADDR;
473                      cnt_image <= cnt_image + 1;
474                  end
475              end
476              else begin
477                  r_ps <= ST_LAYER2_INPUT_ADDR;
478                  addr_start <= addr_start + 1;
479              end
480          end

```

Fig 44

ST_LAYER1_WRITE와 비교하여 한 단계의 if문이 더 추가된 것을 알 수 있다. 가장 마지막 이미지가 입력으로 들어왔을 경우 ST_LAYER3_INPUT_ADDR로 transition하면서, i_rom2_addr, i_ram1_addr, i_ram2_addr를 초기화하고, 아직 마지막 입력이 아닌 경우 다시 ST_LAYER2_INPUT_ADDR로 돌아가고, cnt_image를 1 증가시킨다. 그 외의 동작은 앞선 ST_LAYER2_WRITE와 동일하다.

이제부터는 Fully Connected layer이기 때문에 input address를 계산하는 것이 더 간단하다.

```
486      ST_LAYER3_INPUT_ADDR :  
487      begin  
488          r_ps <= ST_LAYER3_READ;  
489          // ROM  
490          i_rom3_addr <= cnt_rom3;  
491          if(cnt_rom3 == ROM3_ADDR_MAX-1) begin  
492              cnt_rom3 <= 0;  
493          end  
494          else begin  
495              cnt_rom3 <= cnt_rom3 + 1;  
496          end  
497          // RAM  
498          i_ram2_op <= 0;  
499          i_ram2_addr <= cnt_accumulator;  
500      end  
501      ST_LAYER3_READ :  
502      begin  
503          r_ps <= ST_LAYER3_CALCULATE;  
504      end
```

Fig 45

단순히 카운터 하나를 두어 ROM의 address를 계산한다. ROM3_ADDR_MAX-1에 도달하면 0으로 초기화한다. RAM의 경우 accumulator에 몇 번 누적되었는지를 이용하면 된다. 1개의 뉴런에 대한 입력은 5408개(RAM2의 모든 address)를 돌아야 하기 때문에 cnt_accumulator를 address로 가지면 된다.

```
501      ST_LAYER3_READ :  
502      begin  
503          r_ps <= ST_LAYER3_CALCULATE;  
504      end
```

Fig 46

이전과 내용과 동일하다.

```

505      ST_LAYER3_CALCULATE :
506      begin
507          r_ps <= ST_LAYER3_HOLD;
508          i_pe_data <= o_ram2_data;
509          i_pe_weight <= o_rom3_data;
510          i_op_activation <= 0;
511      end

```

Fig 47

이전 내용과 동일하다.

```

512      ST_LAYER3_HOLD:
513      begin
514          i_pe_data <= 0;
515          i_pe_weight <= 0;
516          if(cnt_pe == 1) begin
517              r_ps <= ST_LAYER3_OUTPUT_ADDR;
518              cnt_pe <= 0;
519          end
520          else begin
521              cnt_pe <= cnt_pe + 1;
522          end
523      end

```

Fig 48

이전 내용과 동일하다.

```

524      ST_LAYER3_OUTPUT_ADDR :
525      begin
526          if(cnt_accumulator == RAM2_ADDR_MAX-1) begin
527              r_ps <= ST_LAYER3_WRITE;
528              cnt_accumulator <= 0;
529              i_ram3_op <= 1;
530              i_ram3_addr <= cnt_ram3;
531              i_ram3_data <= o_pe_data;
532              i_accumulator_clr <= 1;
533          end
534          else begin
535              r_ps <= ST_LAYER3_INPUT_ADDR;
536              cnt_accumulator <= cnt_accumulator + 1;
537          end
538      end

```

Fig 49

cnt_accumulator가 RAM2_ADDR_MAX-1(5407)에 도달하면 cnt_accumulator를 초기화하고, RAM3를 write mode로 한다. 동시에 RAM3에 PE의 출력을 넣고, accumulator를 초기화 시킨다.

이때, RAM3의 address에 cnt_ram3를 넣게 되는데 이는 다음 state에서 기술한다. cnt_accumulator가 RAM2_ADDR_MAX-1에 도달하지 못하면 다시 ST_LAYER3_INPUT_ADDR로 돌아간다.

```

539      ST_LAYER3_WRITE :
540      begin
541          i_ram3_op <= 0;
542          i_accumulator_clr <= 0;
543          if(cnt_ram3 == RAM3_ADDR_MAX -1) begin
544              r_ps <= ST_LAYER4_INPUT_ADDR;
545              cnt_ram3 <= 0;
546              i_rom3_addr <= 0;
547              i_ram2_addr <= 0;
548              i_ram3_addr <= 0;
549          end
550          else begin
551              r_ps <= ST_LAYER3_INPUT_ADDR;
552              cnt_ram3 <= cnt_ram3 + 1;
553          end
554      end

```

Fig 50

RAM3를 read mode로 바꾸고, accumulator의 초기화를 종료한다. Cnt_ram3가 RAM3_ADDR_MAX-1(63)에 도달한 경우 ST_LAYER4_INPUT_ADDR로 transition하며 cnt_ram3를 초기화 시키고, i_rom3_addr, i_ram2_addr, i_ram3_addr을 초기화한다. 그렇지 못한 경우 cnt_ram3를 1증가시키고, ST_LAYER3_INPUT_ADDR로 돌아간다.

```

555      ST_LAYER4_INPUT_ADDR :
556      begin
557          r_ps <= ST_LAYER4_READ;
558          i_ram3_op <= 0;
559          i_ram3_addr <= cnt_ram3;
560          i_rom4_addr <= cnt_ram3;
561      end

```

Fig 51

cnt_ram3는 63까지 셀 수 있기 때문에 이를 다시 활용한다. RAM3와 ROM4모두 같은 address를 갖는다.

```

562         ST_LAYER4_READ :
563         begin
564             r_ps <= ST_LAYER4_CALCULATE;
565         end

```

Fig 52

이전 내용과 동일하다.

```

566         ST_LAYER4_CALCULATE :
567         begin
568             r_ps <= ST_LAYER4_HOLD;
569             i_pe_data <= o_ram3_data;
570             i_pe_weight <= o_rom4_data;
571             i_op_activation <= 1;

```

Fig 53

이전 내용과 동일한데, 하나 차이가 있는 것은 i_op_activation을 1로 하며 activation function으로 sigmoid를 선택하는 것이다.

```

573         ST_LAYER4_HOLD :
574         begin
575             i_pe_data <= 0;
576             i_pe_weight <= 0;
577             if(cnt_pe == 5) begin
578                 r_ps <= ST_LAYER4_DECISION;
579                 cnt_pe <= 0;
580             end
581             else begin
582                 cnt_pe <= cnt_pe + 1;
583             end
584         end

```

Fig 54

이전 내용과 동일한데, 하나 차이가 있는 것은 sigmoid를 사용할 경우 PE는 4clk이 증가하기 때문에 cnt_pe를 5까지 세고 transition한다.

```

585 ST_LAYER4_DECISION :
586 begin
587     if(cnt_ram3 == RAM3_ADDR_MAX-1) begin
588         r_ps <= ST_DONE;
589         o_array[0] <= o_decision;
590         o_array[19:1] <= o_array[18:0];
591         done <= 1;
592         cnt_ram3 <= 0;
593         i_ram3_addr <= 0;
594         i_rom4_addr <= 0;
595         i_op_activation <= 0;
596         i_accumulator_clr <= 1;
597     end
598     else begin
599         r_ps <= ST_LAYER4_INPUT_ADDR;
600         cnt_ram3 <= cnt_ram3 + 1;
601     end
602 end

```

Fig 55

cnt_ram3가 RAM3_ADDR_MAX-1(63)에 도달한 경우 ST_DONE으로 transition하며, PE의 o_decision을 o_array에 넣으면서 o_array를 left shift시킨다. Done을 1로 올리고, cnt_ram3, i_ram3_addr, i_rom4_addr를 초기화하고, i_op_activation을 0으로 하면서 accumulator를 초기화한다. cnt_ram3가 RAM3_ADDR_MAX-1에 도달하지 못한 경우 다시 ST_LAYER4_INPUT_ADDR로 돌아가며 cnt_ram3를 1증가시킨다.

```

603 ST_DONE : begin
604     // FOR TEST
605     if(numberOftest == TESTNUMBER -1) begin
606         numberOftest <= 0;
607         r_ps <= ST_DONE;
608     end
609     else begin
610         numberOftest <= numberOftest + 1;
611         r_ps <= ST_IDLE;
612     end

```

Fig 56

numberOftest가 TESTNUMBER-1에 도달하여 모든 test data에 대한 decision이 끝나면 ST_DONE을 유지하며, numberOftest를 초기화한다. Waveform simulation에서는 i_en을 끄는 시점을 control하기 어렵기 때문에 마지막은 ST_DONE에 두어 종료 상태에 위치시키기 위해 사용하였다. 마지막 test data가 decision되지 않은 경우 numberOftest를 1증가시키며 다시 ST_IDLE로 돌아간다.

```

613         done <= 0;
614         i_pe_data <= 0;
615         i_pe_weight <= 0;
616         i_op_activation <= 0;
617         i_accumulator_clr <= 0;
618         i_no_rect_quantize <= 0;
619         i_rom0_addr <= 0;
620         i_ram1_op <= 0;
621         i_ram1_addr <= 0;
622         i_ram1_data <= 0;
623         i_ram2_op <= 0;
624         i_ram2_addr <= 0;
625         i_ram2_data <= 0;
626         i_ram3_op <= 0;
627         i_ram3_addr <= 0;
628         i_ram3_data <= 0;
629         i_rom1_addr <= 0;
630         i_rom2_addr <= 0;
631         i_rom3_addr <= 0;
632         i_rom4_addr <= 0;
633         row <= 0;
634         col <= 0;
635         cnt_row <= 0;
636         cnt_col <= 0;
637         cnt_accumulator <= 0;
638         cnt_pe <= 0;
639         ram_write_addr <= 0;
640         addr_start <= 0;
641         cnt_weight <= 0;
642         cnt_image <= 0;
643         cnt_rom3 <= 0;
644         cnt_ram3 <= 0;
645         end
646     endcase
647 end

```

Fig 57

동시에 모든 register를 초기화한다.

이제 코드에 대한 모든 설명이 끝났다. Memory Base Architecture는 위와 같이 많은 단계는 state를 거쳐야 하며 address를 control하기 위한 complexity가 높다. 그러나, 알고리즘이 정확하게 정해진 후에는 각 단계별로 초기화 등을 정확하게 수행하면 conflict가 발생하지 않도록 설계할 수 있으며, 모든 memory에 대하여 read, write가 되기 때문에 unknown으로 출력되거나 하는 단계가 있다면 문

제가 있는 지점을 파악하는 것이 비교적 수월하다. 단, 이 검증을 위해서는 1회의 decision만 수행하며 RAM을 초기화 시키지 말아야 한다. 모든 단계에서 unknown값이 발생하지 않으면 1회의 decision을 성공적으로 수행하는 것이 되며, 최종적으로 address가 모두 정확하게 변하였는지 하나씩 확인하는 작업이 필요하다. 결과적으로, 우리가 설계한 이번 architecture는 매우 성공적으로 address가 control된다. 기존에 Memory Base Architecture는 FFT에서 주로 사용되었다. FFT의 경우 flow chart상의 모든 연산은 동일한 butterfly를 사용하기 때문에 입력으로 들어오는 data와 twiddle factor를 정확하게 control하여 area와 power를 줄이는 역할을 하였다. 그러나, real-time(on-line)data의 경우 1회의 FFT가 수행되는 동안 많은 latency를 요구하기 때문에 그 latency사이에 들어오는 입력에 대해서는 FFT를 하지 못해 high frequency를 요구하는 경우 사용이 불가능하다는 단점이 존재하였다. 이번 CNN의 경우 일반적인 FFT보다 훨씬 많은 연산을 수행하므로 latency는 FFT와 비교하여도 매우 길다. 따라서 latency사이에 들어오는 입력에 대한 decision을 할 수 없게 된다. 그러나, 우리는 real-time data를 사용하고 있는 것이 아니기 때문에 area를 최소화할 수 있는 이번 module을 설계한 데 있어 이번 프로젝트에서는 이러한 단점을 특별히 걱정해야 할 필요가 없다고 생각한다.

3. Result/Analysis

Latency: i_en이 high가 된 시점부터 done이 high가 되는 시점까지 계산한 1회의 decision에 필요한 clk수는 다음과 같다.

$$Latency = 7699392\ clk$$

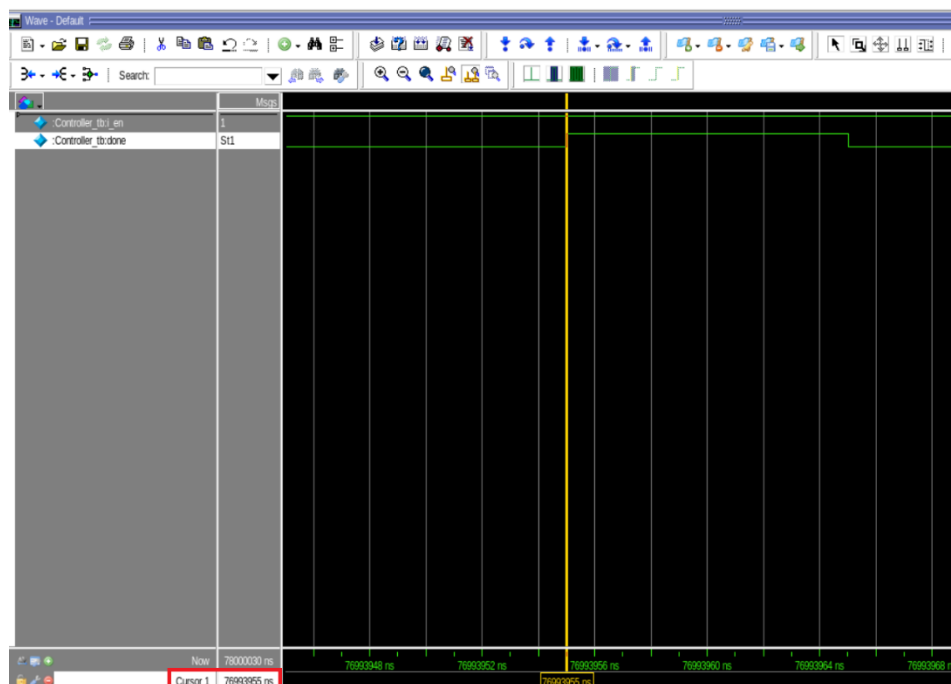


Fig 58

Top인 Controller와 PE로 구성된 Area는 다음과 같다.

```
[3] report_area
Report Instance Areas:
-----+-----+-----+-----+
| Instance | Module | Cells | Cell Area |
-----+-----+-----+-----+
1 | top      |       | 439 | 23874 |
2 | i_PE    | PE    | 7   | 456   |
-----+-----+-----+-----+
```

Fig 59

만약 칩으로 설계될 경우는 패키징 등의 이슈로 실제 칩의 크기는 지금보다 커지게 된다.

다음 Fig 60이 이를 잘 보여준다.

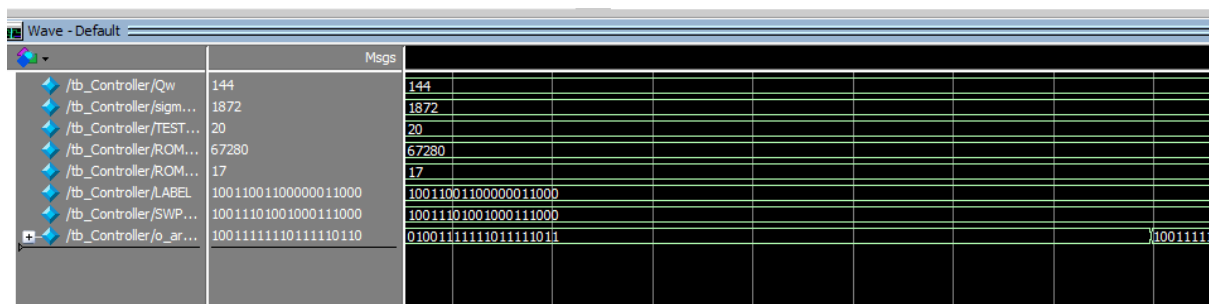
Design Name		Area (squm)	Leakage (uW)
Design Name	Controller		
Total Instances	439	23874	0.087
Macros	0	0	0.000
Pads	0	0	0.000
Phys	0	0	0.000
Blackboxes	0	0	0.000
Cells	439	23874	0.087
Buffers	0	0	0.000
Inverters	42	782	0.004
Clock-Gates	0	0	0.000
Combinational	329	11690	0.038
Latches	0	0	0.000
FlipFlops	68	11402	0.045
Single-Bit FF	68	11402	0.045
Multi-Bit FF	0	0	0.000
Clock-Gated	0		
Bits	68	11402	0.045
Load-Enabled	0		
Clock-Gated	0		
Tristate Pin Count	0		
Physical Info	Placed		
Chip Size (mm x mm)	0.200 x 0.211	42141	
Fixed Cell Area		0	
Phys Only	0	0	
Placeable Area		42141	
Movable Cell Area		23874	
Utilization (%)	56		
Chip Utilization (%)	56		
Total Wire Length (mm)	2.079		
Longest Wire (mm)	0.174		
Average Wire (mm)	0.122		

Fig 60

Chip size는 42141이 되는 것을 알 수 있다. 그러나, 이번 프로젝트는 패키징까지 고려한 사이즈를 묻는 것은 아니라고 판단하여 23874(squm)을 기준으로 하도록 한다. 이때, Controller와 PE만을 측정한 이유는 실제로 칩을 설계할 때, 같은 구조를 가지고 있더라도 weight는 항상 수정될 수 있으며, 실제로는 DRAM등을 banking하여 test data와 weight를 넣게 될 것이기 때문에 위치럼 RAM과 ROM을 제외한 size를 측정하였다.

앞서도 언급하였지만 latency는 극단적으로 많은 clk을 요구한다. 따라서 FPGA에 porting하여 PC에 출력하는 방식이 아닌 waveform을 이용하여 simulation을 하게 되면 20개의 sample을 decision하는데 대략 10분 가까이 되는 시간이 소요되었다. 이러한 모듈의 검증에 있어서는 FPGA를 이용한 simulation이 더 타당할 것으로 생각된다.

우리가 가지고 있는 data set은 test set의 경우 label이 없어 직접 하나씩 눈으로 확인하고, labeling을 해야 했다. 때문에 많은 data set을 사용하지는 못했다. 그러나, Python의 경우 최대 98.7%의 높은 accuracy로 fitting되었을 때, 20개 sample에 대한 prediction이 모두 맞았다. 수 차례의 모델을 돌려서 확인한 결과 평균적으로 1.5개의 miss prediction이 발생하였고, 2개를 초과하는 miss prediction은 없었다. 따라서 20개의 sample만을 이용하더라도 그 결과가 비교적 신뢰가 있다고 생각했고, Verilog의 경우 1회의 검증에 10분이나 소모가 되기 때문에 그 이상을 test하는 것도 어렵다고 판단했다. 20회의 simulation을 수행하였고, 다음은 Python model이 97.46%가 나왔을 때 추출한 parameter와 weight를 이용하여 수행한 결과이다. Python model의 경우 본 경우가 worst case였으나 python model의 accuracy가 높더라도 Verilog에서는 accuracy가 오히려 떨어지는 경우, python model에서 accuracy가 낮음에도 불구하고 Verilog에서는 accuracy가 오히려 높은 경우도 빈번하게 발생하였다.



Signal	Value
/tb_Controller/Qw	144
/tb_Controller/sigm...	1872
/tb_Controller/TEST...	20
/tb_Controller/ROM...	67280
/tb_Controller/ROM...	17
/tb_Controller/LABEL	10011001100000011000
/tb_Controller/SWP...	10011101001000111000
/tb_Controller/o_ar...	1001111110111110110

Fig 61

Samples: 20, Python model accuracy 97.46%

Label: 10011001100000011000

Python: 10011101001000111000 Accuracy: 80%

Verilog: 1001111110111110110 Accuracy: 50%

우리가 측정해야 하는 e-factor는 다음과 같다.

$$e = area * \frac{latency}{accuracy}$$

여기에 area는 23874, latency는 7699392, accuracy는 50을 넣은 e값은 3.68e10이다.

20회의 simulation을 수행하면서 관찰된 Verilog model의 평균 accuracy는 65%였다. 7개를 miss prediction하는 것인데, best case는 5개의 miss prediction, worst case는 10개의 miss prediction이 있었다. 단순히, FP32와 INT8의 차이라고 치부하고 싶은 마음도 있었지만 많은 논문을 찾아본 결과 INT8로 quantization을 하더라도 통상 -1.5%의 accuracy의 loss가 있었다. 우리가 설계한 model은 20개 sample에 대하여 Python은 average 95%, Verilog는 average 65%의 accuracy가 나왔기 때문에 무려 -30%의 loss가 발생했다. 우리의 분석은 다음과 같다.

PTQ(Post Training Quantization)는 activation의 maximum value를 기준으로 Qa(unsigned 8bits quantization factor)를 구하게 된다. 여기서 우리가 간과한 것이 있다. 우리가 설계한 모델을 기준으로 하면 Dense layer이 maximum value는 25정도를 갖지만 2D convolution layer는 보통 2~3의 maximum value를 갖는다. 이때, 25를 기준으로 quantization을 하게 되면, 이렇게 될 경우 2~3은 매우 작기 때문에 2D convolution layer에서 Qa에 의한 loss가 지나치게 많이 발생한다. 또한, shift-add는 0근처에서 매우 큰 loss가 발생한다. 이 때문에 미세하게 dog와 cat을 분류해야 할 때는 error가 커져 버리게 된다. 이를 극복하기 위해서는 sigmoid에 들어오는 입력이 확실하게 큰 값과 작은 값으로 정확하게 분류되어야 한다. 혹은 output layer를 sigmoid가 아닌 ReLu나 Step function등으로 교체해 보는 것도 해결책이 될 수 있다고 생각한다. 그러나, 가장 핵심적인 것은 각 layer의 출력의 maximum value가 큰 차이 없어 고르게 나타나야 quantization을 했을 때, 모든 layer에서 비슷한 수준으로 loss가 발생한다는 것이다. 하나의 layer에 대해서만 loss가 거의 없게 되면 나머지 layer는 모두 신뢰할 수 없게 되기 때문이다. 따라서 모든 activation maximum value의 variance를 측정하여 variance가 작은 모델을 채택해야 하며, 이런 모델을 만들기 위해서는 구조적으로 첫번째 FC에서 적은 횟수의 summation을 수행해야 한다. 즉, FC전에 이미지의 개수와 이미지 사이즈가 충분히 작아져야 variance를 작게 가져갈 수 있게 된다. 그런데, 이러한 상태를 만족하기 위해서는 필연적으로 layer가 깊어져야 하며 깊은 layer는 가장 처음에 밝혔던 것처럼 covariance shift가 일어나 출력 분포가 편향되게 된다. 결국, 깊은 layer와 batch normalization등을 통한 추가적인 module이 필요하게 되며, 대부분의 INT8을 사용하는 NPU논문에서 깊은 layer와 BN을 포함하는 VGG-16, ResNet-50등을 다루는 것이 이 때문이라고 생각한다.

다음 프로젝트에서는 activation의 maximum value뿐만 아니라 variance를 측정하여 충분히 낮은 variance를 갖는 모델을 선택한 후 이번 분석의 타당함을 확인하고자 한다.

4. Source Code

https://github.com/metamong6658/AIIC_PROJECT