Convolutional Neural Network Accelerator for Digit Recognition

Liam Bishop, Daniel Cooke, Felix Hanau, Ryan Kennedy,
Richard Mouradian

May 2022

1 Overview

The goal of our project was to make a Convolutional Neural Network (CNN) accelerator for a Computer Vision Model which will perform digit recognition. For training and testing purposes, the MNIST database was used which contains thousands of images of handwritten digits. Computer Vision models which rely on CNN's are very taxing on general purpose processors because they require several layers of neurons with thousands of parameters and computations. To alleviate this we will leverage the flexibility of the hardware on the FPGA to perform these calculations on the input images in parallel to increase throughput. The CNN starts by receiving an image sent from the operating system on the FPGA. Upon completion of the final layer of computation the CNN will send a probability vector back to the software indicating the decision made by the model.

2 Design

Convolutional Neural Networks (CNN) are a form of Artificial Neural Network that we will be using for image recognition. A CNN is composed of many connected neurons with weights and biases. Each neuron receives input data and performs operations on it using the weights and biases then outputs the transformed data to the next layer. The CNN model we are building is composed of four types of layers: convolution, activation, pooling, and fully connected. These functions are combined to form single layers, then the layers are cascaded to form the CNN.

2.1 The CNN Model

The convolution function is the core function of the CNN and is used to extract features from the input images. The convolution works by taking a small filter or kernel (usually 3x3 or 5x5) and sliding over the input. At each pixel, the dot product is taken of the filter with the corresponding neighborhood of pixels to the current pixel which creates a feature map. Generic kernels exist such as the Sobel filter for detecting vertical and horizontal edges, but our CNN will use unique kernels trained to recognize features specific to our dataset. In this project, Keras was used to obtain weights and bias' for our specific CNN model. Keras is an open source Python library that provides tools for artificial neural networks. For pixels on the edge of the image where the filter would hang over the image can be zero-padded by adding zeros to the edge of the image. The other option would be to ignore the edge pixels and shrink the image after convolution. Our model makes use of convolution in two layers and uses 5x5 kernels along with no zero padding to decrease memory usage. This leads to the size of the input shrinking after each convolution layer.

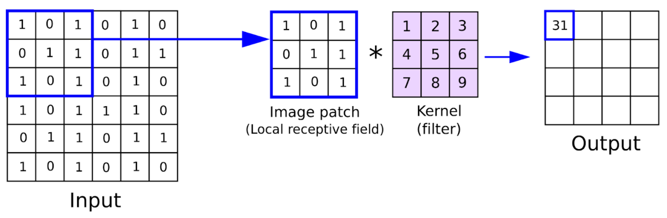


Figure 1: A single step of convolution with a 3x3 kernel.

The activation layer introduces non-linearity into the CNN model and is used to decide if the neuron will "fire" or not. This effect is achieved through the use of an activation function. A naive activation function would be a simple step function on a threshold. Because the networks are trained using the gradient descent method, other activation functions are used that are differentiable and allow for outputs to be less certain. A popular activation function is ReLU which is simply f(x) = max(0,x), it has a low computation cost, is not bounded, and is differentiable over most of the function. Another activation function is sigmoid function:

$$\sigma(x) = \frac{1}{(1+e^{-x})}$$

The sigmoid function is bounded, continuous and differentiable.

The fully connected layer takes as input the flattened matrix of the previous layer's output (ie. 12x4x4 gets flattened to 192x1) and fed as input. The inner workings of this layer take each of these 192 neurons and connect them to each and every neuron in the next layer. The size of this next layer in the fully connected layer can be changed to meet the desired output possibilities as the user wishes. For a next layer of size 10 we have 1920 connections each with a corresponding weight as well as a bias for each neuron in the next layer (10 in size, 1 for each neuron). To illustrate this refer to this image below [4].

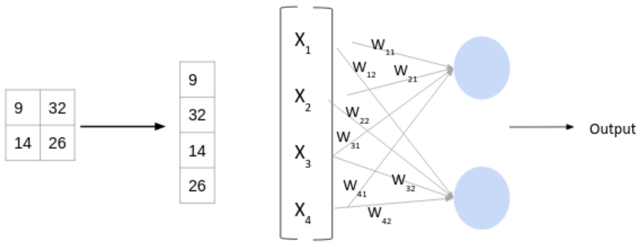


Figure 2: The fully connected layer.

The simplest layer is the pooling layer. The goal for the pooling layer is to reduce the size of the image and thus reduce the number of parameters and overall calculations required. Without the pooling layer the CNN would become very large and slow. Common pooling functions include max pooling and average pooling. In our model both pooling layers pass a 2x2 kernel over the image and output the average value of the four pixels. This reduces the size of the image by $\frac{1}{2}$ in each dimension.

The overall structure of out model is input followed by convolution, pooling, convolution, pooling, fully connected. This is a fairly typical model although many CNNs have far more layers to handle larger images or to obtain higher accuracy. In the convolution layer, our model uses the ReLu activation function. After the fully connected layer, the data is sent to back to the software side, where the sigmoid activation function is used on the data to obtain the final probabilities. This was done in software to avoid having to build a lookup table or function approximation for sigmoid for only 10 values. The diagram below shows the overall structure of the CNN and the size of the inputs and outputs of each layer. It is slightly misleading in that the convolution 2 layer actually has 72 total kernels, 12 for each image from the previous layer. However, the results from each are added together so the actual output is only 12 feature maps. BLOCK DIAGRAM

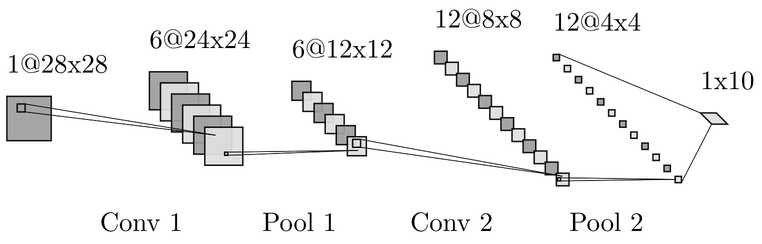


Figure 3: The full CNN.

Figure 4: kernel sizes, output sizes and parameters per layer for 28x28 gray-scale image

Layer Num	Type	Kernel Size	Output Size	Parameters
1	Convolution	6x5x5 (150)	6x24x24 (3456)	156
2	Pooling	6x2x2 (24)	6x12x12 (864)	None
3	Convolution	6x12x5x5 (1800)	12x8x8 (768)	1812
4	Pooling	12x2x2 (48)	12x4x4 (192)	None
5	Fully Connected	None	50	9610

2.2 Software

Even though the accelerator is quite complex, because of the Avalon bus interface the software/hardware interface is relatively simple. The CNN.sv file is the top-level synthesis file and only has 4 input ports that are important for the Linux driver that controls the hardware. The ports are read and write which are wires that are set by ioread and iowrite calls respectively, the other two are readdata and writedata which hold the values being sent to or from the software. Over the Avalon bus, data can be sent from the driver or sent by the accelerator. There are 10 read registers that correspond to the final output of the CNN. There are 4 registers that allow different data to be sent and received from the accelerator. Because the memory that contains the weights and biases is populated using .hex files, the only data that is sent using the driver is the image data. There are two addresses for the image, one for the actual pixel and one for the address corresponding to that pixel. Both are sent as 16 bit integers although the address only uses the first 10 bits and the pixel uses the first 8. The other two registers are for the control signals. One address is for sending control signals, the other is for receiving control signals. These signals interact with the hardware side control state machine. They are again sent as 16 bits but the control signals are only 8-bits wide.

The entire process of classifying an image is contained in the driver program. Calling ioctl with a given input image returns a classification vector to the user. The control of the hardware is also fairly simple. The hardware is notified of an incoming image by sending the signal 1 and the driver waits for the hardware to acknowledge by reading a 1 back. Then the image is transferred over to the hardware sending 1 pixel (16-bits) at a time. From there the driver continues to send increasing numbers to the hardware starting from 2 and not proceeding until it reads the same number back from the hardware. Finally, once signal 6 has been sent and then read back, the driver calls ioread to retrieve the output from the CNN.

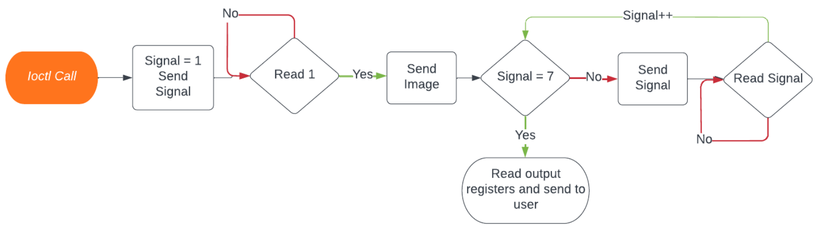


Figure 4: The software control structure

The software program on the user side has two main functions, calling ioctl and preparing the image to be sent. The first thing it does is read in a set amount of images from the test set of the MNIST database. Because all of the math is done in fixed point, the pixel values are also converted to fixed point by multiplying each 8-bit value by 16. This was found to be a good scaling that covered the range required and only resulted in a small decrease in performance when measured on the software only implementation. The software implementation is also built into the main software file and can be used by excluding the "-DUSE_FPGA" compile flag.

2.3 Hardware

The hardware Consists of 4 major regions: the control circuitry, the multiply and accumulate datapath, the pooling modules, and the memory blocks. The control circuitry receives the control signal from the software that dictates which state the hardware should be running in. These states include the load image state where the software sends the input image for processing and the 5 layers that the model consists of. At each state a series of control signals are set according the enable different modules to allow for the proper operations of that state. When the accelerator finishes a layer the return control signal lets the software know to increment to the next state.

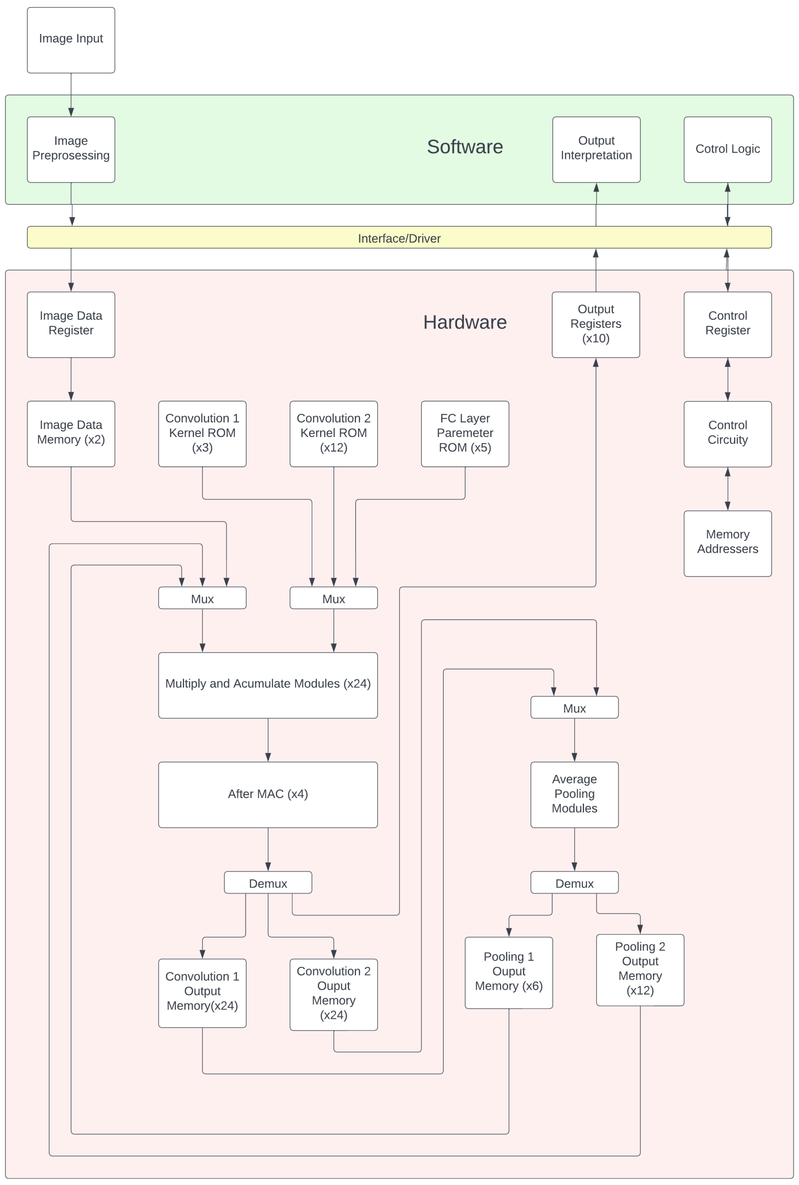
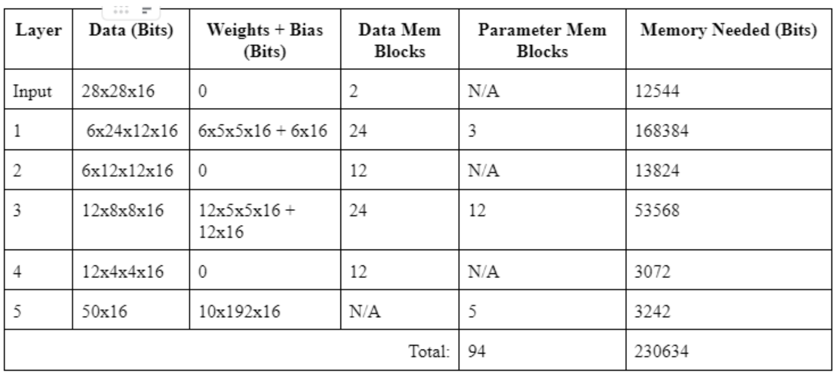


Figure 5: The full CNN block diagram

The multiply and accumulate datapath consists of 24 multiply and accumulate modules, and 4 subsequent modules that preform the additional operations needed based on the current layer. The MAC essentially performs the convolution function by multiplying image data by corresponding weight values and accumulating to get a single final pixel value. This includes adding biases, summing outputs, applying ReLU, and shifting the output to the correct position. Each "after MAC" modules preforms these operations on the output of 6 MAC modules so there is no bottleneck. The pooling modules are quite simple but are important to the model. They perform a 2x2 averaging function by summing all four inputs and returning the sum right shifted by 2.

Finally, and most importantly there are several memory blocks and addressers. The data in between layers is stored in these memory blocks to allow for proper dataflow as well as memory blocks are used to store the parameters for the convolution 1, convolution 2, and fully connected layers. Because each memory block can only be a maximum of dual port the data depending on the layer is either partitioned or redundant memory blocks are used to allow for more than 2 memory accesses simultaneously. These use of multiple memory blocks allows the accelerator to function with a high level of parellelization. The addressers are essentially counters that get activated by enable signals as needed to iterate through the addresses needed to fetch data specific to the current layer or process.

Our systemverilog code was laid out such that all of the modules were contained in the top level CNN.sv file. The modules were then organized into several types. There were math modules that consisted of MAC, pooling, and after_MAC



Layer	Data (Bits)	Weights + Bias (Bits)	Data Mem Blocks	Parameter Mem Blocks	Memory Needed (Bits)
Input	28x28x16	0	2	N/A	12544
1	6x24x12x16	6x5x5x16 + 6x16	24	3	168384
2	6x12x12x16	0	12	N/A	13824
3	12x8x8x16	12x5x5x16 + 12x16	24	12	53568
4	12x4x4x16	0	12	N/A	3072
5	50x16	10x192x16	N/A	5	3242
			Total:	94	230634

Figure 6: Table of the approximate resources required by the CNN model

modules. Then there were the memory counters, RAMs, and ROMs. The ROM folder also contained the .txt files that were used to load in the weights and bias directly to the ROMs. The last type of module is routing which contains the muxes and demuxes necessary to diret the flow of data. All of the data was stored and operated on as 16-bit fixed point values. The size of the memory and their addresses was determined by the number of weights or biases needed to be stored in the particular location.

According to the quartus generated map summary file, our design used 371,328 bits or only about 9% of the total memory. At the start of the project, we were worried about fitting the model on the board, but this proved to not be a concern. The file also stated that 24 DSP blocks were inferred, which is exactly as much as we expected. Lastly, a total of 2534 registers were used. Despite the fact that the design didn't function properly, these values are good metrics and most likely close to the actual values of a working model.

3 Contributions

Richard Mouradian developed the MUX modules and some of the counter modules. He also contributed to the development of the CNN control module. Felix Hanau trained the CNN model and developed the C software implementation. He also implemented several memory counters and developed the SystemVerilog driver for the Verilalator simulation. Liam wrote the convolution function for the software implementation along with writing the device driver, the hardware/software interface through Quartus, and a testbench for the device driver. Ryan worked on the device driver and testbench as well, along with writing some of the software implementation. Daniel Cooke developed the top level CNN module, some of the memory counters, the MAC, after MAC, and the memory blocks with accompanying initialization files for the ROMs. All members contributed to analyzing and debugging the Verilog code in both inspection and using Verilator.

Something that could've been improved upon was the time it took for us to decide what model of CNN we want to put on the boar. Because many of us were not as knowledgeable on the inner workings and overall mechanism of a CNN, we spent a lot of time comparing different possible CNN models to implement. The other concern was choosing a model that would take up more space than what was easily available on the FPGA. The other issue that plagued us was incorrect wire connections because of the sheer number of connections especially in the top-level CNN file. The best advice to future projects is to connect the modules very carefully, as it is easy to accidentally forget to change a connection name, and very difficult to discover the mistake.

4 System Structure

FacialRecognition.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "Parameters.h"
#include "Pool.h"
#include "convolution.h"
#include "fully_connected.h"

struct weights {
  fixed_t conv1_weights[NUM_KERNELS_1 * CONV_KERNEL_SIZE];
  fixed_t conv1_bias[NUM_KERNELS_1];
  fixed_t conv2_weights[NUM_KERNELS_2 * CONV_KERNEL_SIZE];
  fixed_t conv2_bias[NUM_KERNELS_2 * NUM_KERNELS_1 * CONV_KERNEL_SIZE];
  fixed_t fc_weights[NUM_KERNELS_2 * CONV2_OUT_SIZE / POOL_SIZE / POOL_SIZE * NUM_CLASSES];
};

unsigned classify(unsigned char* in_image, struct weights* w) {
  //We should have enough stack space for this on Linux, might just use malloc otherwise.
  //TODO: Is the order of weights in here the same as in keras?
  //TODO: Add code to load weights, input image
  //Some layers assume that their outputs are zeroed out, so do that here.
  fixed_t in_matrix [IMAGE_SIZE];
  fixed_t conv1_out [NUM_KERNELS_1 * CONV1_OUT_SIZE] = {0};
  fixed_t pool1_out [NUM_KERNELS_1 * CONV1_OUT_SIZE / POOL_SIZE / POOL_SIZE] = {0};
  fixed_t conv2_out [NUM_KERNELS_2 * CONV2_OUT_SIZE] = {0};
  fixed_t pool2_out [NUM_KERNELS_2 * CONV2_OUT_SIZE / POOL_SIZE / POOL_SIZE] = {0};
  fixed_t classification_vector[NUM_CLASSES] = {0};

  for (int i = 0; i < IMAGE_SIZE; i++) {
    in_matrix[i] = ((fixed_t)in_image[i]) * FIXED_SCALE;
    if (in_matrix[i] > 0) {
      //fprintf(stderr, "in_matrix [%d]: %d\n", i, in_matrix[i]);
    }
  }
  conv_layer(in_matrix, conv1_out, w->conv1_bias, w->conv1_weights, 1, NUM_KERNELS_1, IMAGE_WIDTH);
  for (int i = 0; i < 24 * 24; i++) {
    if (conv1_out[i] > 0) {
      fprintf(stderr, "conv1 output [%d]: %d\n", i, conv1_out[i]);
    }
  }
  avg_pool(conv1_out, pool1_out, NUM_KERNELS_1 * CONV1_OUT_SIZE, CONV1_OUT_WIDTH);
  for (int i = 0; i < 12 * 12; i++) {
    if (pool1_out[i] > 0) {
      fprintf(stderr, "pool1 output [%d]: %d\n", i % (12 * 12), pool1_out[i]);
    }
  }
  conv_layer(pool1_out, conv2_out, w->conv2_bias, w->conv2_weights, NUM_KERNELS_1, NUM_KERNELS_2, CONV1_OUT_WIDTH / 2);
  for (int i = 0; i < 8 * 8; i++) {
    if (conv2_out[i] > 0) {
      fprintf(stderr, "conv2 output [%d]: %d\n", i, conv2_out[i]);
    }
  }
 avg_pool(conv2_out, pool2_out, NUM_KERNELS_2 * CONV2_OUT_SIZE, CONV2_OUT_WIDTH);
  fully_connected(pool2_out, classification_vector, w->fc_weights, NUM_KERNELS_2 * CONV2_OUT_SIZE / POOL_SIZE / POOL_SIZE, NUM_CLASSES);

  fixed_t max = FIXED_MIN;
  unsigned max_index = 0;
  for (int i = 0; i < NUM_CLASSES; i++) {
    printf("Class %d probability: %.9g\n", i, 1 / (1 + exp(-1.0 * ((double)classification_vector[i]) / FIXED_SCALE)));
    if (classification_vector[i] > max) {
      max = classification_vector[i];
      max_index = i;
    }
  }
  printf("Classified as %u\n", max_index);
  return max_index;
}

#ifdef USE_FPGA
unsigned classify_fpga(int fpga_fd, unsigned char* in_image) {
  cnn_arg_t cnn_io;
  clock_t start, end;
  for (int i = 0; i < IMAGE_SIZE; i++) {
    cnn_io.in_image[i] = ((fixed_t)in_image[i]) * FIXED_SCALE;
  }
  start = clock();
  if (ioctl(fpga_fd, CNN_CLASSIFY, &cnn_io)) {
    perror("ioctl(CNN_CLASSIFY) failed");
    return 0;
  }
  end = clock();
  fprintf(stderr, "time: %f\n\n", (double)(end - start)/CLOCKS_PER_SEC);

  fixed_t max = FIXED_MIN;
  unsigned max_index = 0;
  for (int i = 0; i < NUM_CLASSES; i++) {
    printf("Class %d probability: %.9g\n", i, 1 / (1 + exp(-1.0 * ((double)cnn_io.classification_vector[i]) / FIXED_SCALE)));
    if (cnn_io.classification_vector[i] > max) {
      max = cnn_io.classification_vector[i];
      max_index = i;
    }
  }
  printf("Classified as %u\n", max_index);
  return max_index;
}
#endif

void fill(const char* file, void* dst, unsigned size) {
  FILE* fp = fopen(file, "rb");
  if (!fp) {
    fprintf(stderr, "Could not read %s\n", file);
  }
  fread(dst, sizeof(float), size, fp);
  fclose(fp);
}

void fill_fixed(const char* file, fixed_t* dst, unsigned size) {
  FILE* fp = fopen(file, "rb");
  if (!fp) {
    fprintf(stderr, "Could not read %s\n", file);
  }
  float imm[size];
  fread(imm, sizeof(float), size, fp);
  fclose(fp);
  for (unsigned i = 0; i < size; i++) {
    dst[i] = (fixed_t)roundf(imm[i] * FIXED_SCALE);
  }
}

void write_fixed(const char* file, fixed_t* ptr, unsigned count) {
  FILE* fp = fopen(file, "wb");
  fwrite(ptr, sizeof(signed short), count, fp);
  fclose(fp);
}

int main() {
 clock_t start, end;
#ifdef USE_FPGA
  int fpga_io = open(CNN_IO_FILE,  O_RDWR);
  if (fpga_io == -1) {
    fprintf(stderr, "Could not open CNN I/O file\n");
    return EXIT_FAILURE;
  }
#endif

  struct weights wt;
  fixed_t conv1_weights[CONV_KERNEL_SIZE * NUM_KERNELS_1];
  fixed_t conv2_weights[CONV_KERNEL_SIZE * NUM_KERNELS_1 * NUM_KERNELS_2];
  fill_fixed("../mnist/conv1_weights.bin", conv1_weights, NUM_KERNELS_1 * CONV_KERNEL_SIZE);
  fill_fixed("../mnist/conv1_bias.bin", wt.conv1_bias, NUM_KERNELS_1);
  fill_fixed("../mnist/conv2_weights.bin", conv2_weights, NUM_KERNELS_2 * NUM_KERNELS_1 * CONV_KERNEL_SIZE);
  fill_fixed("../mnist/conv2_bias.bin", wt.conv2_bias, NUM_KERNELS_2);
  fill_fixed("../mnist/fc_weights.bin", wt.fc_weights, NUM_KERNELS_2 * CONV2_OUT_SIZE / POOL_SIZE / POOL_SIZE * NUM_CLASSES);

  //need to transpose some weights, keras puts them in a order that's hard to work with
  for(int out = 0; out < NUM_KERNELS_1; out++) {
    for (int i = 0; i < CONV_KERNEL_SIZE; i++) {
      wt.conv1_weights[out * CONV_KERNEL_SIZE + i] = conv1_weights[(i * NUM_KERNELS_1) + out];
    }
  }
  for(int in = 0; in < NUM_KERNELS_1; in++) {
    for(int out = 0; out < NUM_KERNELS_2; out++) {
      for (int i = 0; i < CONV_KERNEL_SIZE; i++) {
        wt.conv2_weights[(in * NUM_KERNELS_2 + out) * CONV_KERNEL_SIZE + i] = conv2_weights[(i * NUM_KERNELS_2 * NUM_KERNELS_1) + in * NUM_KERNELS_2 + out];
      }
    }
  }
  for (int i = 0; i < 6; i++) {
    fprintf(stderr, "conv1_bias[%d] = %d\n", i, wt.conv1_bias[i]);
  }
  for (int i = 0; i < 25; i++) {
    fprintf(stderr, "conv1 kernel 0 [%d] = %d\n", i, wt.conv1_weights[i]);
  }

  //Should not be needed as long as weights are stable.
#if 0
  write_fixed("../mnist/conv1_weights-16.bin", wt.conv1_weights, NUM_KERNELS_1 * CONV_KERNEL_SIZE);
  write_fixed("../mnist/conv1_bias-16.bin", wt.conv1_bias, NUM_KERNELS_1);
  write_fixed("../mnist/conv2_weights-16.bin", wt.conv2_weights, NUM_KERNELS_1 * NUM_KERNELS_2 * CONV_KERNEL_SIZE);
  write_fixed("../mnist/conv2_bias-16.bin", wt.conv2_bias, NUM_KERNELS_2);
  write_fixed("../mnist/fc_weights-16.bin", wt.fc_weights, NUM_KERNELS_2 * CONV2_OUT_SIZE / POOL_SIZE / POOL_SIZE * NUM_CLASSES);
#endif

#define NUM_IMAGES 32
#define IMAGE_METADATA_OFFSET 16
#define LABEL_METADATA_OFFSET 8
  unsigned char in_image[IMAGE_SIZE * NUM_IMAGES + IMAGE_METADATA_OFFSET] = {0};
  unsigned char in_labels[NUM_IMAGES + LABEL_METADATA_OFFSET] = {0};
  fill("../mnist/t10k-images-idx3-ubyte", (void*)in_image, (NUM_IMAGES * IMAGE_SIZE + IMAGE_METADATA_OFFSET) / sizeof(float));
  fill("../mnist/t10k-labels-idx1-ubyte", (void*)in_labels, (NUM_IMAGES + LABEL_METADATA_OFFSET) / sizeof(float));
  unsigned correct_classifications = 0;
  for (int i = 0; i < 1; i++) {
#ifdef USE_FPGA
    unsigned prediction = classify_fpga(fpga_io, &(in_image[i * IMAGE_SIZE + IMAGE_METADATA_OFFSET]));
#else
    start = clock();
    unsigned prediction = classify(&(in_image[i * IMAGE_SIZE + IMAGE_METADATA_OFFSET]), &wt);
    end = clock();
    fprintf(stderr, "time: %f\n\n", (double)(end - start)/CLOCKS_PER_SEC);
#endif
    printf("Actual label: %u\n", in_labels[i + LABEL_METADATA_OFFSET]);
    correct_classifications += (in_labels[i + LABEL_METADATA_OFFSET] == prediction);
  }
  fprintf(stderr, "Correct classifications: %u / %u\n", correct_classifications, 1);

#ifdef USE_FPGA
  close(fpga_io);
#endif
  return 0;
}


#include "math.h"

/*
* src: the 1-d flattened array of all image features
* dst: the array neurons in the next layer for which we are calculating
* weights: the weights of each connection (if the previous layer is
size 192 and the next layer is 10 this weight vector is of length 1920)
* bias: the bias of each neuron in the next layer (for a size 10 fully connected
layer this is of size 10)
*/

void fully_connected(fixed_t *src, fixed_t *dst, const fixed_t *weights, int src_size, int dst_size)
{
    //TODO: Make the number of in images a parameter instead of a magic number
    //TODO: Need to transform order of the input data as the weights expect a different order.
    //TODO: might be somewhat inaccurate due to float type
    for (int dest_index = 0; dest_index < (dst_size); dest_index++) {
      for (int image = 0; image < 12; image++) {
        for (int pos = 0; pos < src_size / 12; pos++) {
          dst[dest_index] += ((weights[(pos * (12) + image) * dst_size + dest_index] * src[image * (src_size / 12) + pos]) >> FIXED_SCALE_LOG);
        }
      }
    }

    //Bias, sigmoid not needed on FPGA. We don't use bias, sigmoid is optional and may
    //be done in software for prettier classification values in console output.
}

#endif


#ifndef _CONVOLUTION_H_
#define _CONVOLUTION_H_

#include "Parameters.h"

#define MAX(a, b) (a > b ? a : b)

/*
 * src: memory location for previously computed images
 * dst: memory location for newly processed images
 * bias: Location for bias, value added to each pixel of an image
 * weights: memory location for filters
 * num_src: the number of images from the previous layer
 * num_dst: the number of images that will be produced from this layer
 * src_size: the size of each image from the previous layer C1 = 28
 *
 */
void conv_layer(fixed_t *src, fixed_t *dst, const fixed_t *bias, const fixed_t *weights, int num_src, int num_dst, int src_size)
{
    int dst_size =  src_size - 4;
    int dst_index, src_index;

    for(int in = 0; in < num_src; in++)
    {
        // The image that will be convolved
        const fixed_t *image = &src[in * src_size * src_size];

        for(int out = 0; out < num_dst; out++)
        {
            // Pointer to where to store the output image
            fixed_t *output = &dst[out * dst_size * dst_size];

            // Weight filter for this image
            // This calculation assumes that the weight vector only contains weights for
            // this specific layer.
            const fixed_t *weight = &weights[(in * num_dst + out) * CONV_KERNEL_SIZE];

            // Matrix dotsum
            for(int i = 2; i < src_size - 2; i++)
            {
                for(int j = 2; j < src_size - 2; j++)
                {
                    // Index for output value
                    dst_index = ((i - 2)*dst_size) + j - 2;

                    // starting index for source value
                    src_index = ((i - 2)*src_size) + j - 2;
                    output[dst_index] +=
                        //Row 1
                        ((image[src_index] * weight[0] + image[src_index + 1] * weight[1] + image[src_index + 2] * weight[2] + image[src_index + 3] * weight[3] + image[src_index +

                        //Row 2
                        image[src_index + src_size] * weight[5] + image[src_index + src_size + 1] * weight[6] + image[src_index + src_size + 2] * weight[7] +
                        image[src_index + src_size + 3] * weight[8] + image[src_index + src_size + 4] * weight[9] +

                        //Row 3
                        image[src_index + 2 * src_size] * weight[10] + image[src_index + 2 * src_size + 1] * weight[11] + image[src_index + 2 * src_size + 2] * weight[12] +
                        image[src_index + 2 * src_size + 3] * weight[13] + image[src_index + 2 * src_size + 4] * weight[14] +

                        //Row 4
                        image[src_index + 3 * src_size] * weight[15] + image[src_index + 3 * src_size + 1] * weight[16] + image[src_index + 3 * src_size + 2] * weight[17] +
                        image[src_index + 3 * src_size + 3] * weight[18] + image[src_index + 3 * src_size + 4] * weight[19] +

                        //Row 5
                        image[src_index + 4 * src_size] * weight[20] + image[src_index + 4 * src_size + 1] * weight[21] + image[src_index + 4 * src_size + 2] * weight[22] +
                        image[src_index + 4 * src_size + 3] * weight[23] + image[src_index + 4 * src_size + 4] * weight[24])
                        //Doing multiplication, need to divide by scaling factor twice
                        >> FIXED_SCALE_LOG);
                    if (in == 0 && out == 0) {
                      //fprintf(stderr, "pre-bias output %d %d: %d\n", i, j, output[dst_index]);
                    }
                }
            }
        }
    }

    //This loop is needed when num_src > 1, since in this case output[dst_index] will be modified several times and we need to do ReLU on the final values
    for(int out = 0; out < num_dst; out++) {
        // Pointer to where to store the output image
        fixed_t *output = &dst[out * dst_size * dst_size];
        // Bias value for this image
        fixed_t bi = bias[out];
        for (int i = 0; i < dst_size * dst_size; i++) {
            // Activation (ReLU function), easier to implement than sigmoid
            output[i] = MAX(0, output[i] + bi);
            if (out == 0) {
              //fprintf(stderr, "post-bias output %d: %d\n", i, output[i]);
            }
        }
    }
}

#endif // _CONVOLUTION_H_


#ifndef POOL_H
#define POOL_H
#include <string.h>
#include "Parameters.h"

//max pooling and fixed point support are not needed at this time.
#if 0
//TODO: Maybe we'll switch to just integers, but plan with fixed point for
//now. This header will need to define the fixed point type and some
//functions. Otherwise just use int or perhaps unsigned char types.
#include "FixedPoint.h"

#define POOL_SIZE 2

//TODO: Consider https://graphics.stanford.edu/~seander/bithacks.html#IntegerMinOrMax for branchless max operation in hardware

//This will compile and work correctly as long as fixed_t is an alias for
//an integer type
#define MAX(a, b) (a > b ? a : b)
//TODO: Is pooling 3D or just some of the other layers? Switch to
//fixed_t** in that case and add outer loop for layers
//TODO: Could return the output if memory allocation is allowed in
//here, handle it elsewhere for now.
void max_pool(fixed_t* in, fixed_t* out) {
  //DATA_WIDTH_POOL needs to be defined in Parameters.h
  const int out_width = DATA_WIDTH_POOL / POOL_SIZE;

  for (int i = 0; i < out_width; i++) {
    for (int j = 0; j < out_width; j++) {
      //TODO: If there are negative values we may need to adjust this
      //and the max operation.
      fixed_t max = 0;
      //Helps de-clutter the in array access
      int start_idx = POOL_SIZE * (i * DATA_WIDTH_POOL + j);
      for (int k = 0; k < POOL_SIZE; k++) {
        for (int l = 0; l < POOL_SIZE; l++) {
          max = MAX(max, in[start_idx + k * DATA_WIDTH_POOL + l]);
        }
      }
      out[i * out_width + j] = max;
    }
  }
  //Could alternatively do this, which is simpler but would be slow
  //with a non po2 pool size due to division.
  /*memset(out, 0, out_width * out_width * sizeof(fixed_t));
  for (int i = 0; i < DATA_WIDTH_POOL; i++) {
    for (int j = 0; j < DATA_WIDTH_POOL; j++) {
      out[(i / DATA_WIDTH_POOL) * out_width + (j / DATA_WIDTH_POOL)] =
        MAX(out[(i / DATA_WIDTH_POOL) * out_width + (j / DATA_WIDTH_POOL)],         in[i * DATA_WIDTH_POOL + j]);
    }
  }*/
}

//TODO: Could also develop a struct based interface instead where
//parameters are passed to struct, e.g.
//Pool layer4(3);
//...
//fixed_t* pooled_data = layer4.process(relu_data);
#endif

#define POOL_SIZE 2

//Need src_row_size to know where to read information from consecutive rows. Number of channels is not needed
void avg_pool(fixed_t* in, fixed_t* out, int src_size, int src_row_size) {
  for (int i = 0; i < src_size / src_row_size / POOL_SIZE; i++) {
    for (int j = 0; j < src_row_size / POOL_SIZE; j++) {
      fixed_t sum = 0;
      for (int k = 0; k < POOL_SIZE; k++) {
        for (int l = 0; l < POOL_SIZE; l++) {
          sum += in[(i * POOL_SIZE + k) * (src_row_size) + j * POOL_SIZE + l];
          if (in[(i * POOL_SIZE + k) * (src_row_size) + j * POOL_SIZE + l] > 0) {
            fprintf(stderr, "pool %d %d: %d\n", i * (src_row_size / POOL_SIZE) + j, (i * POOL_SIZE + k) * (src_row_size) + j * POOL_SIZE + l, in[(i * POOL_SIZE + k) * (src_row_siz
          }
        }
      }
      out[i * (src_row_size / POOL_SIZE) + j] = sum / POOL_SIZE / POOL_SIZE;
    }
  }
}

#endif


#ifndef PARAMETER_H
#define PARAMETER_H

#ifdef __KERNEL__
#include <linux/ioctl.h>
#include <linux/limits.h>
#else
#include <limits.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#endif

#define FIXED_POINT_SIZE
#define IMAGE_WIDTH 28
#define IMAGE_SIZE (IMAGE_WIDTH * IMAGE_WIDTH)
#define POOL_SIZE 2
#define CONV_KERNEL_SIZE (5 * 5)

#define NUM_CLASSES 10
#define NUM_KERNELS_1 6
#define NUM_KERNELS_2 12
#define CONV1_OUT_WIDTH (IMAGE_WIDTH - 4)
#define CONV1_OUT_SIZE (CONV1_OUT_WIDTH * CONV1_OUT_WIDTH)
#define CONV2_OUT_WIDTH (CONV1_OUT_WIDTH / POOL_SIZE - 4)
#define CONV2_OUT_SIZE (CONV2_OUT_WIDTH * CONV2_OUT_WIDTH)
```

```
typedef short fixed_t;
#define FIXED_SCALE_LOG 4
#define FIXED_SCALE (1 << FIXED_SCALE_LOG)
#define FIXED_MIN (fixed_t)(SHRT_MIN)

typedef struct {
  fixed_t in_image[IMAGE_SIZE];
  fixed_t classification_vector[NUM_CLASSES];
} cnn_arg_t;

#define CNN_IO_FILE "/dev/cnn"
#define CNN_DRIVER_MAGIC 'q'
#define CNN_CLASSIFY _IOWR(CNN_DRIVER_MAGIC, 1, cnn_arg_t*)
#endif


#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#include "../software-testbench/Parameters.h"

#define DRIVER_NAME "cnn"
#define CONTROL_IN_REG(x) (x)
#define CONTROL_OUT_REG(x) ((x) + 2)
#define INPUT_ADDR_REG(x) ((x) + 4)
#define INPUT_DATA_REG(x) ((x) + 6)
#define OUTPUT_REG_BASE(x) ((x) + 8)

struct cnn_dev {
    struct resource res; // Registers
    void __iomem *virtbase; // Location of registers in memory
    cnn_arg_t data;
} dev;

static void send_image(fixed_t *image)
{
    fixed_t i;

    for (i = 0; i < IMAGE_SIZE; i++){
        iowrite16(i, INPUT_ADDR_REG(dev.virtbase));
        iowrite16(image[i], INPUT_DATA_REG(dev.virtbase));
    }
}

static void control(fixed_t *image) {
  fixed_t control_in;
  fixed_t control_out;

  // Load Image
  control_in = 1;
  iowrite16(control_in, CONTROL_IN_REG(dev.virtbase));
  do {
    control_out = ioread16(CONTROL_OUT_REG(dev.virtbase));
  } while (control_out != control_in);
  pr_info("load_image");

  send_image(image);

  control_in = 2;
  pr_info("load_image");
  for (; control_in < 7; control_in++) {
    // Send signal to start next layer
    iowrite16(control_in, CONTROL_IN_REG(dev.virtbase));

    // Wait for done signal from hardware
    do {
      control_out = ioread16(CONTROL_OUT_REG(dev.virtbase));
    } while (control_out != control_in);
  }
}

static void read_output(fixed_t *vector)
{
    int i;

    for (i = 0; i < NUM_CLASSES; i++){
        vector[i] = ioread16(OUTPUT_REG_BASE(dev.virtbase) + (2 * i));
        vector[i] = ioread16(OUTPUT_REG_BASE(dev.virtbase) + (2 * i));
    }
}

static long cnn_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    cnn_arg_t in_data;

    switch (cmd)
    {
    case CNN_CLASSIFY:
            // Copy image from user
        if (copy_from_user(&in_data, (cnn_arg_t *)arg, sizeof(cnn_arg_t)))
            return -EACCES;

        pr_info("CNN_CLASSIFY");
        control(in_data.in_image);

            // Read output
            read_output(in_data.classification_vector);
        if (copy_to_user((cnn_arg_t *)arg, &in_data, sizeof(cnn_arg_t)))
            return -EACCES;
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

static const struct file_operations cnn_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = cnn_ioctl,
};

static struct miscdevice cnn_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &cnn_fops,
};

static int __init cnn_probe(struct platform_device *pdev)
{
    int ret;

    pr_info("Before Misc register");
    /* Register as misc device, also creates /dev/cnn/ */
    ret = misc_register(&cnn_misc_device);
    pr_info("Misc register");
    // Obtain address of the registers from device tree
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if(ret)
    {
        ret = -ENOENT;
        goto out_deregister;
    }
    pr_info("of address");

    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
      ret = -EBUSY;
      goto out_deregister;
    }
    pr_info("request mem region");


    // Arrange access to registers
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if(dev.virtbase ==  NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }
    pr_info("of iomap");

    return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&cnn_misc_device);
    return ret;
}

static int cnn_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&cnn_misc_device);
    return 0;
}

#ifdef CONFIG_OF
static const struct of_device_id cnn_of_match[] = {
    {.compatible = "csee4840,cnn_driver-1.0"},
    {},
};
MODULE_DEVICE_TABLE(of, cnn_of_match);
#endif

static struct platform_driver cnn_driver = {
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(cnn_of_match),
    },
    .remove = __exit_p(cnn_remove),
};

static int __init cnn_driver_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&cnn_driver, cnn_probe);
}

static void __exit cnn_driver_exit(void)
{
    platform_driver_unregister(&cnn_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(cnn_driver_init);
module_exit(cnn_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Liam Bishop and Ryan Kennedy");
MODULE_DESCRIPTION("CNN driver for fpga");


module CNN(input logic clk, reset, write, read, chipselect,
           input logic [15:0] writedata,
           input logic [3:0] address,
           output logic [15:0] readdata);

    logic signed [15:0] img_data;
    logic [9:0] img_mem_addr_write;
    logic [7:0] ctrl;
    always_ff @(posedge clk) begin
        if(chipselect == 1'b1 && (write == 1'b1 || read == 1'b1)) begin
            case (address)
                4'h0 : ctrl <= writedata[7:0];
                4'h1 : readdata <= {8'b00000000, return_ctrl};
                4'h2 : img_mem_addr_write <= writedata[9:0];
                4'h3 : img_data <= writedata;
                4'h4 : readdata <= result_0;
                4'h5 : readdata <= result_1;
                4'h6 : readdata <= result_2;
                4'h7 : readdata <= result_3;
                4'h8 : readdata <= result_4;
                4'h9 : readdata <= result_5;
                4'ha : readdata <= result_6;
                4'hb : readdata <= result_7;
                4'hc : readdata <= result_8;
                4'hd : readdata <= result_9;
                default : readdata <= 0;
            endcase
        end
    end

logic [1:0] MAC_layer;
logic [7:0] return_ctrl;
logic pooling_layer, img_load, rMAC, MAC_enable, Conv1_layer, P1_layer, Conv2_layer, P2_layer, FC_layer,
      img_mem_read_reset, conv1_mem_write_reset, conv1_mem_read_reset, conv1_k_mem_read_reset, P1_mem_read_reset, P1_mem_write_reset,
      conv2_mem_write_reset, conv2_mem_read_reset, conv2_k_mem_read_reset, P2_mem_write_reset, P2_mem_read_reset, fc_mem_read_reset,
      img_mem_read_done, conv1_mem_write_done, conv1_mem_read_done, conv1_k_mem_read_done, P1_mem_read_done, P1_mem_write_done,
      conv2_mem_write_done, conv2_mem_read_done, conv2_k_mem_read_done, P2_mem_write_done, P2_mem_read_done, fc_mem_read_done;

CNN_ctrl CNN_ctrl(.*);

//------------------------------------------------------------------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------------------------------------------------------------------------
//
//  IMAGE DATA MEMORY
//
//------------------------------------------------------------------------------------------------------------------------------------------------------

logic signed [15:0] img_mem_0_q_a, img_mem_0_q_b, img_mem_1_q_a, img_mem_1_q_b;
logic [9:0] img_mem_addr0_read, img_mem_addr1_read, img_mem_addr2_read, img_mem_addr3_read, img_mem_addrw0r, img_mem_addrw2r;

img_mem_read img_mem_read(.clk(clk), .reset(img_mem_read_reset), .enable(Conv1_layer), .addr0(img_mem_addr0_read), .addr1(img_mem_addr1_read), .addr2(img_mem_addr2_read), .addr3(
mux_2to1 #(10) img_mem_addr_mux_0(.data_in_0(img_mem_addr_write), .data_in_1(img_mem_addr0_read), .sel(Conv1_layer), .data_out(img_mem_addrw0r));
mux_2to1 #(10) img_mem_addr_mux_1(.data_in_0(img_mem_addr_write), .data_in_1(img_mem_addr2_read), .sel(Conv1_layer), .data_out(img_mem_addrw2r));

//image memory. They are redundant to allow for 4 accesses
img_mem img_mem_0(.address_a(img_mem_addrw0r), .address_b(img_mem_addr1_read), .clock(clk), .data_a(img_data), .data_b(16'b0000000000000000), .wren_a(img_load), .wren_b(1'b0), .q
img_mem img_mem_1(.address_a(img_mem_addrw2r), .address_b(img_mem_addr3_read), .clock(clk), .data_a(img_data), .data_b(16'b0000000000000000), .wren_a(img_load), .wren_b(1'b0), .q_

//----------------------------------------------------------------------------------------------------------------------------------------
//
//  CONV1 OUTPUT MEMORY
//
//----------------------------------------------------------------------------------------------------------------------------------------

logic signed [15:0] conv1_mem_0_0_data_a, conv1_mem_0_0_data_b, conv1_mem_0_1_data_a, conv1_mem_0_1_data_b,
                    conv1_mem_1_0_data_a, conv1_mem_1_0_data_b, conv1_mem_1_1_data_a, conv1_mem_1_1_data_b,
                    conv1_mem_2_0_data_a, conv1_mem_2_0_data_b, conv1_mem_2_1_data_a, conv1_mem_2_1_data_b,
                    conv1_mem_3_0_data_a, conv1_mem_3_0_data_b, conv1_mem_3_1_data_a, conv1_mem_3_1_data_b,
                    conv1_mem_4_0_data_a, conv1_mem_4_0_data_b, conv1_mem_4_1_data_a, conv1_mem_4_1_data_b,
                    conv1_mem_5_0_data_a, conv1_mem_5_0_data_b, conv1_mem_5_1_data_a, conv1_mem_5_1_data_b,
                    conv1_mem_0_0_q_a, conv1_mem_0_0_q_b, conv1_mem_0_1_q_a, conv1_mem_0_1_q_b, conv1_mem_0_2_q_a, conv1_mem_0_2_q_b, conv1_mem_0_3_q_a, conv1_mem_0_3_q_b,
                    conv1_mem_1_0_q_a, conv1_mem_1_0_q_b, conv1_mem_1_1_q_a, conv1_mem_1_1_q_b, conv1_mem_1_2_q_a, conv1_mem_1_2_q_b, conv1_mem_1_3_q_a, conv1_mem_1_3_q_b,
                    conv1_mem_2_0_q_a, conv1_mem_2_0_q_b, conv1_mem_2_1_q_a, conv1_mem_2_1_q_b, conv1_mem_2_2_q_a, conv1_mem_2_2_q_b, conv1_mem_2_3_q_a, conv1_mem_2_3_q_b,
                    conv1_mem_3_0_q_a, conv1_mem_3_0_q_b, conv1_mem_3_1_q_a, conv1_mem_3_1_q_b, conv1_mem_3_2_q_a, conv1_mem_3_2_q_b, conv1_mem_3_3_q_a, conv1_mem_3_3_q_b,
                    conv1_mem_4_0_q_a, conv1_mem_4_0_q_b, conv1_mem_4_1_q_a, conv1_mem_4_1_q_b, conv1_mem_4_2_q_a, conv1_mem_4_2_q_b, conv1_mem_4_3_q_a, conv1_mem_4_3_q_b,
                    conv1_mem_5_0_q_a, conv1_mem_5_0_q_b, conv1_mem_5_1_q_a, conv1_mem_5_1_q_b, conv1_mem_5_2_q_a, conv1_mem_5_2_q_b, conv1_mem_5_3_q_a, conv1_mem_5_3_q_b;
logic [8:0] conv1_addr0_write, conv1_addr1_write, conv1_addr0_read, conv1_addr1_read, conv1_addr2_read, conv1_addr3_read, conv1_addr0w0r, conv1_addr0w2r, conv1_addr1w1r, conv1_add

conv1_mem_write conv1_mem_write(.clk(clk), .reset(conv1_mem_write_reset), .enable(Conv1_layer), .addr0(conv1_addr0_write), .addr1(conv1_addr1_write), .done(conv1_mem_write_done));
conv1_mem_read conv1_mem_read(.clk(clk), .reset(conv1_mem_read_reset), .enable(P1_layer), .addr0(conv1_addr0_read), .addr1(conv1_addr1_read), .addr2(conv1_addr2_read), .addr3(conv
mux_2to1 #(9) conv1_mem_addr_mux_0(.data_in_0(conv1_addr0_write), .data_in_1(conv1_addr0_read), .sel(P1_layer), .data_out(conv1_addr0w0r));
mux_2to1 #(9) conv1_mem_addr_mux_1(.data_in_0(conv1_addr0_write), .data_in_1(conv1_addr2_read), .sel(P1_layer), .data_out(conv1_addr0w2r));
mux_2to1 #(9) conv1_mem_addr_mux_2(.data_in_0(conv1_addr1_write), .data_in_1(conv1_addr1_read), .sel(P1_layer), .data_out(conv1_addr1w1r));
mux_2to1 #(9) conv1_mem_addr_mux_3(.data_in_0(conv1_addr1_write), .data_in_1(conv1_addr3_read), .sel(P1_layer), .data_out(conv1_addr1w3r));

//Conv1 output image 0. Each stores half of the image to allow for 8 accesses. _0 _2 and _1 _3 are redundant
conv1_mem conv1_mem_0_0(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_0_0_data_a), .data_b(conv1_mem_0_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_0_1(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_0_1_data_a), .data_b(conv1_mem_0_1_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_0_2(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_0_0_data_a), .data_b(conv1_mem_0_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_0_3(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_0_1_data_a), .data_b(conv1_mem_0_1_data_b), .wren_a(Conv1_layer), .w
//Conv1 output image 1. Each stores half of the image to allow for 8 accesses. _0 _2 and _1 _3 are redundant
conv1_mem conv1_mem_1_0(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_1_0_data_a), .data_b(conv1_mem_1_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_1_1(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_1_1_data_a), .data_b(conv1_mem_1_1_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_1_2(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_1_0_data_a), .data_b(conv1_mem_1_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_1_3(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_1_1_data_a), .data_b(conv1_mem_1_1_data_b), .wren_a(Conv1_layer), .w
//Conv1 output image 2. Each stores half of the image to allow for 8 accesses. _0 _2 and _1 _3 are redundant
conv1_mem conv1_mem_2_0(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_2_0_data_a), .data_b(conv1_mem_2_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_2_1(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_2_1_data_a), .data_b(conv1_mem_2_1_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_2_2(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_2_0_data_a), .data_b(conv1_mem_2_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_2_3(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_2_1_data_a), .data_b(conv1_mem_2_1_data_b), .wren_a(Conv1_layer), .w
//Conv1 output image 3. Each stores half of the image to allow for 8 accesses. _0 _2 and _1 _3 are redundant
conv1_mem conv1_mem_3_0(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_3_0_data_a), .data_b(conv1_mem_3_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_3_1(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_3_1_data_a), .data_b(conv1_mem_3_1_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_3_2(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_3_0_data_a), .data_b(conv1_mem_3_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_3_3(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_3_1_data_a), .data_b(conv1_mem_3_1_data_b), .wren_a(Conv1_layer), .w
//Conv1 output image 4. Each stores half of the image to allow for 8 accesses. _0 _2 and _1 _3 are redundant
conv1_mem conv1_mem_4_0(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_4_0_data_a), .data_b(conv1_mem_4_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_4_1(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_4_1_data_a), .data_b(conv1_mem_4_1_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_4_2(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_4_0_data_a), .data_b(conv1_mem_4_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_4_3(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_4_1_data_a), .data_b(conv1_mem_4_1_data_b), .wren_a(Conv1_layer), .w
//Conv1 output image 5. Each stores half of the image to allow for 8 accesses. _0 _2 and _1 _3 are redundant
conv1_mem conv1_mem_5_0(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_5_0_data_a), .data_b(conv1_mem_5_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_5_1(.address_a(conv1_addr0w0r), .address_b(conv1_addr1w1r), .clock(clk), .data_a(conv1_mem_5_1_data_a), .data_b(conv1_mem_5_1_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_5_2(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_5_0_data_a), .data_b(conv1_mem_5_0_data_b), .wren_a(Conv1_layer), .w
conv1_mem conv1_mem_5_3(.address_a(conv1_addr0w2r), .address_b(conv1_addr1w3r), .clock(clk), .data_a(conv1_mem_5_1_data_a), .data_b(conv1_mem_5_1_data_b), .wren_a(Conv1_layer), .w

//----------------------------------------------------------------------------------------------------------------------------------------
//
//  CONV1 KERNEL MEMORY
//
//----------------------------------------------------------------------------------------------------------------------------------------

logic signed [15:0] conv1_k_g0_mem_q_a, conv1_k_g0_mem_q_b, conv1_k_g1_mem_q_a, conv1_k_g1_mem_q_b, conv1_k_g2_mem_q_a, conv1_k_g2_mem_q_b;
logic[5:0] conv1_k_mem_addr0_read, conv1_k_mem_addr1_read;

conv1_k_mem_read conv1_k_mem_read(.clk(clk), .reset(conv1_k_mem_read_reset), .enable(Conv1_layer), .addr0(conv1_k_mem_addr0_read), .addr1(conv1_k_mem_addr1_read), .done(conv1_k_me

//Memories for conv1 kernels. Each holds 2 25x25 kernels.
conv1_k_g0_mem conv1_k_g0_mem(.address_a(conv1_k_mem_addr0_read), .address_b(conv1_k_mem_addr1_read), .clock(clk), .q_a(conv1_k_g0_mem_q_a), .q_b(conv1_k_g0_mem_q_b));
conv1_k_g1_mem conv1_k_g1_mem(.address_a(conv1_k_mem_addr0_read), .address_b(conv1_k_mem_addr1_read), .clock(clk), .q_a(conv1_k_g1_mem_q_a), .q_b(conv1_k_g1_mem_q_b));
conv1_k_g2_mem conv1_k_g2_mem(.address_a(conv1_k_mem_addr0_read), .address_b(conv1_k_mem_addr1_read), .clock(clk), .q_a(conv1_k_g2_mem_q_a), .q_b(conv1_k_g2_mem_q_b));

//----------------------------------------------------------------------------------------------------------------------------------------
//
//  P1 OUTPUT MEMORY
//
//----------------------------------------------------------------------------------------------------------------------------------------

logic [7:0] p1_addr_0_read, p1_addr_0_write, p1_addr_1_write, p1_addr0w0r;
mux_2to1 #(8) p1_addr_mux_0(.data_in_0(p1_addr_0_write), .data_in_1(p1_addr_0_read), .sel(Conv2_layer), .data_out(p1_addr0w0r)); //mux to combine write and read addresses
P1_mem_read P1_mem_read(.clk(clk), .reset(P1_mem_read_reset), .enable(Conv2_layer), .addr0(p1_addr_0_read), .done(P1_mem_read_done));
P1_mem_write P1_mem_write(.clk(clk), .reset(P1_mem_write_reset), .enable(P1_layer), .addr0(p1_addr_0_write), .addr1(p1_addr_1_write), .done(P1_mem_write_done));

logic signed [15:0] p1_mem_0_data_a, p1_mem_0_data_b, p1_mem_1_data_a, p1_mem_1_data_b,
                    p1_mem_2_data_a, p1_mem_2_data_b, p1_mem_3_data_a, p1_mem_3_data_b,
                    p1_mem_4_data_a, p1_mem_4_data_b, p1_mem_5_data_a, p1_mem_5_data_b,
                    p1_mem_0_q_a, p1_mem_1_q_a, p1_mem_2_q_a, p1_mem_3_q_a, p1_mem_4_q_a, p1_mem_5_q_a;
//Memories for the 6 outputs of p1. Each is unique
p1_mem p1_mem_0(.address_a(p1_addr0w0r), .address_b(p1_addr_1_write), .clock(clk), .data_a(p1_mem_0_data_a), .data_b(p1_mem_0_data_b), .wren_a(P1_layer), .wren_b(P1_layer), .q_a(p
p1_mem p1_mem_1(.address_a(p1_addr0w0r), .address_b(p1_addr_1_write), .clock(clk), .data_a(p1_mem_1_data_a), .data_b(p1_mem_1_data_b), .wren_a(P1_layer), .wren_b(P1_layer), .q_a(p
p1_mem p1_mem_2(.address_a(p1_addr0w0r), .address_b(p1_addr_1_write), .clock(clk), .data_a(p1_mem_2_data_a), .data_b(p1_mem_2_data_b), .wren_a(P1_layer), .wren_b(P1_layer), .q_a(p
p1_mem p1_mem_3(.address_a(p1_addr0w0r), .address_b(p1_addr_1_write), .clock(clk), .data_a(p1_mem_3_data_a), .data_b(p1_mem_3_data_b), .wren_a(P1_layer), .wren_b(P1_layer), .q_a(p
p1_mem p1_mem_4(.address_a(p1_addr0w0r), .address_b(p1_addr_1_write), .clock(clk), .data_a(p1_mem_4_data_a), .data_b(p1_mem_4_data_b), .wren_a(P1_layer), .wren_b(P1_layer), .q_a(p
p1_mem p1_mem_5(.address_a(p1_addr0w0r), .address_b(p1_addr_1_write), .clock(clk), .data_a(p1_mem_5_data_a), .data_b(p1_mem_5_data_b), .wren_a(P1_layer), .wren_b(P1_layer), .q_a(p

//----------------------------------------------------------------------------------------------------------------------------------------
//
//  CONV2 OUTPUT MEMORY
//
//----------------------------------------------------------------------------------------------------------------------------------------

logic [5:0] conv2_addr0_write, conv2_addr0_read, conv2_addr1_read, conv2_addr2_read, conv2_addr3_read, conv2_addr0w0r, conv2_addr0w2r;
logic [1:0] conv2_count;
conv2_mem_write conv2_mem_write(.clk(clk), .reset(conv2_mem_write_reset), .enable(Conv2_layer), .addr0(conv2_addr0_write), .count(conv2_count), .done(conv2_mem_write_done));
conv2_mem_read conv2_mem_read(.clk(clk), .reset(conv2_mem_read_reset), .enable(P2_layer), .addr0(conv2_addr0_read), .addr1(conv2_addr1_read), .addr2(conv2_addr2_read), .addr3(conv
mux_2to1 #(6) conv2_mem_addr_mux_0(.data_in_0(conv2_addr0_write), .data_in_1(conv2_addr0_read), .sel(P2_layer), .data_out(conv2_addr0w0r));
mux_2to1 #(6) conv2_mem_addr_mux_1(.data_in_0(conv2_addr0_write), .data_in_1(conv2_addr2_read), .sel(P2_layer), .data_out(conv2_addr0w2r));

logic signed [15:0] conv2_mem_0_data, conv2_mem_1_data, conv2_mem_2_data, conv2_mem_3_data,
                    conv2_mem_4_data, conv2_mem_5_data, conv2_mem_6_data, conv2_mem_7_data,
                    conv2_mem_8_data, conv2_mem_9_data, conv2_mem_10_data, conv2_mem_11_data,
                    conv2_mem_0_0_q_a, conv2_mem_0_0_q_b, conv2_mem_0_1_q_a, conv2_mem_0_1_q_b,
                    conv2_mem_1_0_q_a, conv2_mem_1_0_q_b, conv2_mem_1_1_q_a, conv2_mem_1_1_q_b,
                    conv2_mem_2_0_q_a, conv2_mem_2_0_q_b, conv2_mem_2_1_q_a, conv2_mem_2_1_q_b,
                    conv2_mem_3_0_q_a, conv2_mem_3_0_q_b, conv2_mem_3_1_q_a, conv2_mem_3_1_q_b,
                    conv2_mem_4_0_q_a, conv2_mem_4_0_q_b, conv2_mem_4_1_q_a, conv2_mem_4_1_q_b,
                    conv2_mem_5_0_q_a, conv2_mem_5_0_q_b, conv2_mem_5_1_q_a, conv2_mem_5_1_q_b,
                    conv2_mem_6_0_q_a, conv2_mem_6_0_q_b, conv2_mem_6_1_q_a, conv2_mem_6_1_q_b,
                    conv2_mem_7_0_q_a, conv2_mem_7_0_q_b, conv2_mem_7_1_q_a, conv2_mem_7_1_q_b,
                    conv2_mem_8_0_q_a, conv2_mem_8_0_q_b, conv2_mem_8_1_q_a, conv2_mem_8_1_q_b,
                    conv2_mem_9_0_q_a, conv2_mem_9_0_q_b, conv2_mem_9_1_q_a, conv2_mem_9_1_q_b,
                    conv2_mem_10_0_q_a, conv2_mem_10_0_q_b, conv2_mem_10_1_q_a, conv2_mem_10_1_q_b,
                    conv2_mem_11_0_q_a, conv2_mem_11_0_q_b, conv2_mem_11_1_q_a, conv2_mem_11_1_q_b;

//Conv2 output 0. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_0_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_0_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
conv2_mem conv2_mem_0_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_0_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
//Conv2 output 1. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_1_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_1_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
conv2_mem conv2_mem_1_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_1_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
//Conv2 output 2. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_2_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_2_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
conv2_mem conv2_mem_2_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_2_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
//Conv2 output 3. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_3_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_3_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
conv2_mem conv2_mem_3_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_3_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
//Conv2 output 4. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_4_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_4_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
conv2_mem conv2_mem_4_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_4_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
//Conv2 output 5. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_5_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_5_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
conv2_mem conv2_mem_5_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_5_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
//Conv2 output 6. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_6_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_6_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
conv2_mem conv2_mem_6_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_6_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
//Conv2 output 7. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_7_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_7_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
conv2_mem conv2_mem_7_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_7_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
//Conv2 output 8. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_8_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_8_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
conv2_mem conv2_mem_8_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_8_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
//Conv2 output 9. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_9_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_9_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
conv2_mem conv2_mem_9_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_9_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .wre
//Conv2 output 10. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_10_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_10_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .w
conv2_mem conv2_mem_10_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_10_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .w
//Conv2 output 11. They are redundant to allow for 4 accesses
conv2_mem conv2_mem_11_0(.address_a(conv2_addr0w0r), .address_b(conv2_addr1_read), .clock(clk), .data_a(conv2_mem_11_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .w
conv2_mem conv2_mem_11_1(.address_a(conv2_addr0w2r), .address_b(conv2_addr3_read), .clock(clk), .data_a(conv2_mem_11_data), .data_b(16'b0000000000000000), .wren_a(Conv2_layer), .w

//------------------------------------------------------------------------------------------------------------------------------------------------------
//
//  CONV2 KERNEL MEMORY
//
//------------------------------------------------------------------------------------------------------------------------------------------------------

logic[7:0] conv2_k_mem_addr0_read, conv2_k_mem_addr1_read;
conv2_k_mem_read conv2_k_mem_read(.clk(clk), .reset(conv2_k_mem_read_reset), .enable(Conv2_layer), .addr0(conv2_k_mem_addr0_read), .addr1(conv2_k_mem_addr1_read), .done(conv2_k_me

logic signed [15:0] conv2_k_g0_mem_q_a, conv2_k_g0_mem_q_b, conv2_k_g1_mem_q_a, conv2_k_g1_mem_q_b,
                    conv2_k_g2_mem_q_a, conv2_k_g2_mem_q_b, conv2_k_g3_mem_q_a, conv2_k_g3_mem_q_b,
                    conv2_k_g4_mem_q_a, conv2_k_g4_mem_q_b, conv2_k_g5_mem_q_a, conv2_k_g5_mem_q_b,
                    conv2_k_g6_mem_q_a, conv2_k_g6_mem_q_b, conv2_k_g7_mem_q_a, conv2_k_g7_mem_q_b,
                    conv2_k_g8_mem_q_a, conv2_k_g8_mem_q_b, conv2_k_g9_mem_q_a, conv2_k_g9_mem_q_b,
                    conv2_k_g10_mem_q_a, conv2_k_g10_mem_q_b, conv2_k_g11_mem_q_a, conv2_k_g11_mem_q_b;
//Memories for conv2 kernels. Each holds 6 25x25 kernels.
conv2_k_g0_mem conv2_k_g0_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g0_mem_q_a), .q_b(conv2_k_g0_mem_q_b));
conv2_k_g1_mem conv2_k_g1_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g1_mem_q_a), .q_b(conv2_k_g1_mem_q_b));
conv2_k_g2_mem conv2_k_g2_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g2_mem_q_a), .q_b(conv2_k_g2_mem_q_b));
conv2_k_g3_mem conv2_k_g3_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g3_mem_q_a), .q_b(conv2_k_g3_mem_q_b));
conv2_k_g4_mem conv2_k_g4_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g4_mem_q_a), .q_b(conv2_k_g4_mem_q_b));
conv2_k_g5_mem conv2_k_g5_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g5_mem_q_a), .q_b(conv2_k_g5_mem_q_b));
conv2_k_g6_mem conv2_k_g6_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g6_mem_q_a), .q_b(conv2_k_g6_mem_q_b));
conv2_k_g7_mem conv2_k_g7_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g7_mem_q_a), .q_b(conv2_k_g7_mem_q_b));
conv2_k_g8_mem conv2_k_g8_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g8_mem_q_a), .q_b(conv2_k_g8_mem_q_b));
conv2_k_g9_mem conv2_k_g9_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g9_mem_q_a), .q_b(conv2_k_g9_mem_q_b));
conv2_k_g10_mem conv2_k_g10_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g10_mem_q_a), .q_b(conv2_k_g10_mem_q_b));
conv2_k_g11_mem conv2_k_g11_mem(.address_a(conv2_k_mem_addr0_read), .address_b(conv2_k_mem_addr1_read), .clock(clk), .q_a(conv2_k_g11_mem_q_a), .q_b(conv2_k_g11_mem_q_b));

//------------------------------------------------------------------------------------------------------------------------------------------------------
//
//  P2 OUTPUT MEMORY
//
//------------------------------------------------------------------------------------------------------------------------------------------------------

//wires that go from P2 memories/addressers
logic [3:0] P2_mem_sel; //wire that will control the muxes that make sure the P2 memory is read squentially block by block
logic signed [15:0] P2_mem_0_data, P2_mem_1_data, P2_mem_2_data, P2_mem_3_data, P2_mem_4_data, P2_mem_5_data,
                    P2_mem_6_data, P2_mem_7_data, P2_mem_8_data, P2_mem_9_data, P2_mem_10_data, P2_mem_11_data,
                    P2_mem_0_q, P2_mem_1_q, P2_mem_2_q, P2_mem_3_q, P2_mem_4_q, P2_mem_5_q,
                    P2_mem_6_q, P2_mem_7_q, P2_mem_8_q, P2_mem_9_q, P2_mem_10_q, P2_mem_11_q; //q lines for p2 memory block
logic [3:0] P2_addr0_write, P2_addr0_read, P2_addr0w0r; //wires for addressing

//addressers for writing and reading the P2 memories
P2_mem_write P2_mem_write(.clk(clk), .reset(P2_mem_write_reset), .enable(P2_layer), .addr0(P2_addr0_write), .done(P2_mem_write_done));
P2_mem_read P2_mem_read(.clk(clk), .reset(P2_mem_read_reset), .enable(FC_layer), .addr0(P2_addr0_read), .count(P2_mem_sel), .done(P2_mem_read_done));
mux_2to1 #(4) P2_addr_mux_0(.data_in_0(P2_addr0_write), .data_in_1(P2_addr0_read), .sel(FC_layer), .data_out(P2_addr0w0r)); //mux to combine write and read addresses

//Memories for the 12 outputs of p2. Each is unique
p2_mem p2_mem_0(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_0_data), .wren(P2_layer), .q(P2_mem_0_q));
p2_mem p2_mem_1(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_1_data), .wren(P2_layer), .q(P2_mem_1_q));
p2_mem p2_mem_2(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_2_data), .wren(P2_layer), .q(P2_mem_2_q));
p2_mem p2_mem_3(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_3_data), .wren(P2_layer), .q(P2_mem_3_q));
p2_mem p2_mem_4(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_4_data), .wren(P2_layer), .q(P2_mem_4_q));
p2_mem p2_mem_5(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_5_data), .wren(P2_layer), .q(P2_mem_5_q));
p2_mem p2_mem_6(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_6_data), .wren(P2_layer), .q(P2_mem_6_q));
p2_mem p2_mem_7(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_7_data), .wren(P2_layer), .q(P2_mem_7_q));
p2_mem p2_mem_8(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_8_data), .wren(P2_layer), .q(P2_mem_8_q));
p2_mem p2_mem_9(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_9_data), .wren(P2_layer), .q(P2_mem_9_q));
p2_mem p2_mem_10(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_10_data), .wren(P2_layer), .q(P2_mem_10_q));
p2_mem p2_mem_11(.address(P2_addr0w0r), .clock(clk), .data(P2_mem_11_data), .wren(P2_layer), .q(P2_mem_11_q));

//------------------------------------------------------------------------------------------------------------------------------------------------------
//
//  FC WEIGHT MEMORY
//
//------------------------------------------------------------------------------------------------------------------------------------------------------

//wires for FC memories
logic signed [15:0] fc_g0_mem_q_a, fc_g0_mem_q_b, fc_g1_mem_q_a, fc_g1_mem_q_b, fc_g2_mem_q_a, fc_g2_mem_q_b, fc_g3_mem_q_a, fc_g3_mem_q_b, fc_g4_mem_q_a, fc_g4_mem_q_b;
logic [8:0] fc_mem_addr0_read, fc_mem_addr1_read;

//addressers for writing and reading the FC weight memories
fc_mem_read fc_mem_read(.clk(clk), .reset(fc_mem_read_reset), .enable(FC_layer), .addr0(fc_mem_addr0_read), .addr1(fc_mem_addr1_read), .done(fc_mem_read_done));

//Memories for FC weights. Each holds 2 neurons's 192 weights(384 in total)
fc_g0_mem fc_g0_mem(.address_a(fc_mem_addr0_read), .address_b(fc_mem_addr1_read), .clock(clk), .q_a(fc_g0_mem_q_a), .q_b(fc_g0_mem_q_b));
fc_g1_mem fc_g1_mem(.address_a(fc_mem_addr0_read), .address_b(fc_mem_addr1_read), .clock(clk), .q_a(fc_g1_mem_q_a), .q_b(fc_g1_mem_q_b));
fc_g2_mem fc_g2_mem(.address_a(fc_mem_addr0_read), .address_b(fc_mem_addr1_read), .clock(clk), .q_a(fc_g2_mem_q_a), .q_b(fc_g2_mem_q_b));
fc_g3_mem fc_g3_mem(.address_a(fc_mem_addr0_read), .address_b(fc_mem_addr1_read), .clock(clk), .q_a(fc_g3_mem_q_a), .q_b(fc_g3_mem_q_b));
fc_g4_mem fc_g4_mem(.address_a(fc_mem_addr0_read), .address_b(fc_mem_addr1_read), .clock(clk), .q_a(fc_g4_mem_q_a), .q_b(fc_g4_mem_q_b));

//------------------------------------------------------------------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------------------------------------------------------------------------
//
//  MAC
//
//------------------------------------------------------------------------------------------------------------------------------------------------------

//signals for input to MACs and to connect the MACs to the after MACs
logic signed [15:0] MAC_in_data_0, MAC_in_data_1, MAC_in_data_2, MAC_in_data_3, MAC_in_data_4, MAC_in_data_5,
                    MAC_in_data_6, MAC_in_data_7, MAC_in_data_8, MAC_in_data_9, MAC_in_data_10, MAC_in_data_11,
                    MAC_in_data_12, MAC_in_data_13, MAC_in_data_14, MAC_in_data_15, MAC_in_data_16, MAC_in_data_17,
                    MAC_in_data_18, MAC_in_data_19, MAC_in_data_20, MAC_in_data_21, MAC_in_data_22, MAC_in_data_23,
                    MAC_in_para_0, MAC_in_para_1, MAC_in_para_2, MAC_in_para_3, MAC_in_para_4, MAC_in_para_5,
                    MAC_in_para_6, MAC_in_para_7, MAC_in_para_8, MAC_in_para_9, MAC_in_para_10, MAC_in_para_11,
                    MAC_in_para_12, MAC_in_para_13, MAC_in_para_14, MAC_in_para_15, MAC_in_para_16, MAC_in_para_17,
                    MAC_in_para_18, MAC_in_para_19, MAC_in_para_20, MAC_in_para_21, MAC_in_para_22, MAC_in_para_23,
                    FC_in_data_0,  FC_in_data_1,  FC_in_data_2,  FC_in_data_3,  FC_in_data_4,  FC_in_data_5,
                    FC_in_data_6,  FC_in_data_7,  FC_in_data_8,  FC_in_data_9,  FC_in_data_10,  FC_in_data_11,
                    FC_in_data_12,  FC_in_data_13,  FC_in_data_14,  FC_in_data_15,  FC_in_data_16,  FC_in_data_17,
                    FC_in_data_18,  FC_in_data_19,  FC_in_data_20,  FC_in_data_21,  FC_in_data_22,  FC_in_data_23,
                    out_conv2_0, out_conv2_1, out_conv2_2, out_conv2_3,
                    after_MAC_0_out_0, after_MAC_0_out_1, after_MAC_0_out_2, after_MAC_0_out_3, after_MAC_0_out_4,
                    after_MAC_0_out_5, after_MAC_1_out_0, after_MAC_1_out_1, after_MAC_1_out_2, after_MAC_1_out_3;
logic signed [31:0] conv2_bias_0, conv2_bias_1, conv2_bias_2, conv2_bias_3,
                    MAC_out_0, MAC_out_1, MAC_out_2, MAC_out_3, MAC_out_4, MAC_out_5,
                    MAC_out_6, MAC_out_7, MAC_out_8, MAC_out_9, MAC_out_10, MAC_out_11,
                    MAC_out_12, MAC_out_13, MAC_out_14, MAC_out_15, MAC_out_16, MAC_out_17,
                    MAC_out_18, MAC_out_19, MAC_out_20, MAC_out_21, MAC_out_22, MAC_out_23;
```

```
//Data muxes for FC layer. Responsible for making sure each input is read sequentially by all MACs
mux_12to1 data_mux_12to1_0(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_0));
mux_12to1 data_mux_12to1_1(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_1));
mux_12to1 data_mux_12to1_2(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_2));
mux_12to1 data_mux_12to1_3(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_3));
mux_12to1 data_mux_12to1_4(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_4));
mux_12to1 data_mux_12to1_5(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_5));
mux_12to1 data_mux_12to1_6(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_6));
mux_12to1 data_mux_12to1_7(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_7));
mux_12to1 data_mux_12to1_8(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_8));
mux_12to1 data_mux_12to1_9(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_9));
mux_12to1 data_mux_12to1_10(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                           .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                           .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                           .sel(P2_mem_sel), .data_out(FC_in_data_10));
mux_12to1 data_mux_12to1_11(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_11));
mux_12to1 data_mux_12to1_12(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_12));
mux_12to1 data_mux_12to1_13(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_13));
mux_12to1 data_mux_12to1_14(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_14));
mux_12to1 data_mux_12to1_15(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_15));
mux_12to1 data_mux_12to1_16(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_16));
mux_12to1 data_mux_12to1_17(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_17));
mux_12to1 data_mux_12to1_18(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_18));
mux_12to1 data_mux_12to1_19(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_19));
mux_12to1 data_mux_12to1_20(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_20));
mux_12to1 data_mux_12to1_21(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_21));
mux_12to1 data_mux_12to1_22(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_22));
mux_12to1 data_mux_12to1_23(.data_in_0(P2_mem_0_q), .data_in_1(P2_mem_1_q), .data_in_2(P2_mem_2_q), .data_in_3(P2_mem_3_q),
                            .data_in_4(P2_mem_4_q), .data_in_5(P2_mem_5_q), .data_in_6(P2_mem_6_q), .data_in_7(P2_mem_7_q),
                            .data_in_8(P2_mem_8_q), .data_in_9(P2_mem_9_q), .data_in_10(P2_mem_10_q), .data_in_11(P2_mem_11_q),
                            .sel(P2_mem_sel), .data_out(FC_in_data_23));

//Muxes to control the inputs to the MACs based on layer
mux_3to1 data_mux_3to1_0(.data_in_0(img_mem_0_q_a), .data_in_1(p1_mem_0_q_a), .data_in_2(FC_in_data_0), .sel(MAC_layer), .data_out(MAC_in_data_0));
mux_3to1 data_mux_3to1_1(.data_in_0(img_mem_0_q_a), .data_in_1(p1_mem_1_q_a), .data_in_2(FC_in_data_1), .sel(MAC_layer), .data_out(MAC_in_data_1));
mux_3to1 data_mux_3to1_2(.data_in_0(img_mem_0_q_a), .data_in_1(p1_mem_2_q_a), .data_in_2(FC_in_data_2), .sel(MAC_layer), .data_out(MAC_in_data_2));
mux_3to1 data_mux_3to1_3(.data_in_0(img_mem_0_q_a), .data_in_1(p1_mem_3_q_a), .data_in_2(FC_in_data_3), .sel(MAC_layer), .data_out(MAC_in_data_3));
mux_3to1 data_mux_3to1_4(.data_in_0(img_mem_0_q_a), .data_in_1(p1_mem_4_q_a), .data_in_2(FC_in_data_4), .sel(MAC_layer), .data_out(MAC_in_data_4));
mux_3to1 data_mux_3to1_5(.data_in_0(img_mem_0_q_a), .data_in_1(p1_mem_5_q_a), .data_in_2(FC_in_data_5), .sel(MAC_layer), .data_out(MAC_in_data_5));
mux_3to1 data_mux_3to1_6(.data_in_0(img_mem_0_q_b), .data_in_1(p1_mem_0_q_a), .data_in_2(FC_in_data_6), .sel(MAC_layer), .data_out(MAC_in_data_6));
mux_3to1 data_mux_3to1_7(.data_in_0(img_mem_0_q_b), .data_in_1(p1_mem_1_q_a), .data_in_2(FC_in_data_7), .sel(MAC_layer), .data_out(MAC_in_data_7));
mux_3to1 data_mux_3to1_8(.data_in_0(img_mem_0_q_b), .data_in_1(p1_mem_2_q_a), .data_in_2(FC_in_data_8), .sel(MAC_layer), .data_out(MAC_in_data_8));
mux_3to1 data_mux_3to1_9(.data_in_0(img_mem_0_q_b), .data_in_1(p1_mem_3_q_a), .data_in_2(FC_in_data_9), .sel(MAC_layer), .data_out(MAC_in_data_9));
mux_3to1 data_mux_3to1_10(.data_in_0(img_mem_0_q_b), .data_in_1(p1_mem_4_q_a), .data_in_2(FC_in_data_10), .sel(MAC_layer), .data_out(MAC_in_data_10));
mux_3to1 data_mux_3to1_11(.data_in_0(img_mem_0_q_b), .data_in_1(p1_mem_5_q_a), .data_in_2(FC_in_data_11), .sel(MAC_layer), .data_out(MAC_in_data_11));
mux_3to1 data_mux_3to1_12(.data_in_0(img_mem_1_q_a), .data_in_1(p1_mem_0_q_a), .data_in_2(FC_in_data_12), .sel(MAC_layer), .data_out(MAC_in_data_12));
mux_3to1 data_mux_3to1_13(.data_in_0(img_mem_1_q_a), .data_in_1(p1_mem_1_q_a), .data_in_2(FC_in_data_13), .sel(MAC_layer), .data_out(MAC_in_data_13));
mux_3to1 data_mux_3to1_14(.data_in_0(img_mem_1_q_a), .data_in_1(p1_mem_2_q_a), .data_in_2(FC_in_data_14), .sel(MAC_layer), .data_out(MAC_in_data_14));
mux_3to1 data_mux_3to1_15(.data_in_0(img_mem_1_q_a), .data_in_1(p1_mem_3_q_a), .data_in_2(FC_in_data_15), .sel(MAC_layer), .data_out(MAC_in_data_15));
mux_3to1 data_mux_3to1_16(.data_in_0(img_mem_1_q_a), .data_in_1(p1_mem_4_q_a), .data_in_2(FC_in_data_16), .sel(MAC_layer), .data_out(MAC_in_data_16));
mux_3to1 data_mux_3to1_17(.data_in_0(img_mem_1_q_a), .data_in_1(p1_mem_5_q_a), .data_in_2(FC_in_data_17), .sel(MAC_layer), .data_out(MAC_in_data_17));
mux_3to1 data_mux_3to1_18(.data_in_0(img_mem_1_q_b), .data_in_1(p1_mem_0_q_a), .data_in_2(FC_in_data_18), .sel(MAC_layer), .data_out(MAC_in_data_18));
mux_3to1 data_mux_3to1_19(.data_in_0(img_mem_1_q_b), .data_in_1(p1_mem_1_q_a), .data_in_2(FC_in_data_19), .sel(MAC_layer), .data_out(MAC_in_data_19));
mux_3to1 data_mux_3to1_20(.data_in_0(img_mem_1_q_b), .data_in_1(p1_mem_2_q_a), .data_in_2(FC_in_data_20), .sel(MAC_layer), .data_out(MAC_in_data_20));
mux_3to1 data_mux_3to1_21(.data_in_0(img_mem_1_q_b), .data_in_1(p1_mem_3_q_a), .data_in_2(FC_in_data_21), .sel(MAC_layer), .data_out(MAC_in_data_21));
mux_3to1 data_mux_3to1_22(.data_in_0(img_mem_1_q_b), .data_in_1(p1_mem_4_q_a), .data_in_2(FC_in_data_22), .sel(MAC_layer), .data_out(MAC_in_data_22));
mux_3to1 data_mux_3to1_23(.data_in_0(img_mem_1_q_b), .data_in_1(p1_mem_5_q_a), .data_in_2(FC_in_data_23), .sel(MAC_layer), .data_out(MAC_in_data_23));

//Muxes to make sure proper input parameter based on the correct ROM. 1 for conv1, 1 for conv2, 1 for fc.
mux_3to1 para_mux_3to1_0(.data_in_0(conv1_k_g0_mem_q_a), .data_in_1(conv2_k_g0_mem_q_a), .data_in_2(fc_g0_mem_q_a), .sel(MAC_layer), .data_out(MAC_in_para_0));
mux_3to1 para_mux_3to1_1(.data_in_0(conv1_k_g0_mem_q_b), .data_in_1(conv2_k_g0_mem_q_b), .data_in_2(fc_g0_mem_q_b), .sel(MAC_layer), .data_out(MAC_in_para_1));
mux_3to1 para_mux_3to1_2(.data_in_0(conv1_k_g1_mem_q_a), .data_in_1(conv2_k_g1_mem_q_a), .data_in_2(fc_g1_mem_q_a), .sel(MAC_layer), .data_out(MAC_in_para_2));
mux_3to1 para_mux_3to1_3(.data_in_0(conv1_k_g1_mem_q_b), .data_in_1(conv2_k_g1_mem_q_b), .data_in_2(fc_g1_mem_q_b), .sel(MAC_layer), .data_out(MAC_in_para_3));
mux_3to1 para_mux_3to1_4(.data_in_0(conv1_k_g2_mem_q_a), .data_in_1(conv2_k_g2_mem_q_a), .data_in_2(fc_g2_mem_q_a), .sel(MAC_layer), .data_out(MAC_in_para_4));
mux_3to1 para_mux_3to1_5(.data_in_0(conv1_k_g2_mem_q_b), .data_in_1(conv2_k_g2_mem_q_b), .data_in_2(fc_g2_mem_q_b), .sel(MAC_layer), .data_out(MAC_in_para_5));
mux_3to1 para_mux_3to1_6(.data_in_0(conv1_k_g0_mem_q_a), .data_in_1(conv2_k_g3_mem_q_a), .data_in_2(fc_g3_mem_q_a), .sel(MAC_layer), .data_out(MAC_in_para_6));
mux_3to1 para_mux_3to1_7(.data_in_0(conv1_k_g0_mem_q_b), .data_in_1(conv2_k_g3_mem_q_b), .data_in_2(fc_g3_mem_q_b), .sel(MAC_layer), .data_out(MAC_in_para_7));
mux_3to1 para_mux_3to1_8(.data_in_0(conv1_k_g1_mem_q_a), .data_in_1(conv2_k_g4_mem_q_a), .data_in_2(fc_g4_mem_q_a), .sel(MAC_layer), .data_out(MAC_in_para_8));
mux_3to1 para_mux_3to1_9(.data_in_0(conv1_k_g1_mem_q_b), .data_in_1(conv2_k_g4_mem_q_b), .data_in_2(fc_g4_mem_q_b), .sel(MAC_layer), .data_out(MAC_in_para_9));
mux_3to1 para_mux_3to1_10(.data_in_0(conv1_k_g2_mem_q_a), .data_in_1(conv2_k_g5_mem_q_a), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_10));
mux_3to1 para_mux_3to1_11(.data_in_0(conv1_k_g2_mem_q_b), .data_in_1(conv2_k_g5_mem_q_b), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_11));
mux_3to1 para_mux_3to1_12(.data_in_0(conv1_k_g0_mem_q_a), .data_in_1(conv2_k_g6_mem_q_a), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_12));
```

```
mux_3to1 para_mux_3to1_13(.data_in_0(conv1_k_g0_mem_q_b), .data_in_1(conv2_k_g6_mem_q_b), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_13));
mux_3to1 para_mux_3to1_14(.data_in_0(conv1_k_g1_mem_q_a), .data_in_1(conv2_k_g7_mem_q_a), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_14));
mux_3to1 para_mux_3to1_15(.data_in_0(conv1_k_g1_mem_q_b), .data_in_1(conv2_k_g7_mem_q_b), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_15));
mux_3to1 para_mux_3to1_16(.data_in_0(conv1_k_g2_mem_q_a), .data_in_1(conv2_k_g8_mem_q_a), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_16));
mux_3to1 para_mux_3to1_17(.data_in_0(conv1_k_g2_mem_q_b), .data_in_1(conv2_k_g8_mem_q_b), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_17));
mux_3to1 para_mux_3to1_18(.data_in_0(conv1_k_g0_mem_q_a), .data_in_1(conv2_k_g9_mem_q_a), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_18));
mux_3to1 para_mux_3to1_19(.data_in_0(conv1_k_g0_mem_q_b), .data_in_1(conv2_k_g9_mem_q_b), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_19));
mux_3to1 para_mux_3to1_20(.data_in_0(conv1_k_g1_mem_q_a), .data_in_1(conv2_k_g10_mem_q_a), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_20));
mux_3to1 para_mux_3to1_21(.data_in_0(conv1_k_g1_mem_q_b), .data_in_1(conv2_k_g10_mem_q_b), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_21));
mux_3to1 para_mux_3to1_22(.data_in_0(conv1_k_g2_mem_q_a), .data_in_1(conv2_k_g11_mem_q_a), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_22));
mux_3to1 para_mux_3to1_23(.data_in_0(conv1_k_g2_mem_q_b), .data_in_1(conv2_k_g11_mem_q_b), .data_in_2(16'b0000000000000000), .sel(MAC_layer), .data_out(MAC_in_para_23));

//bias muxes
mux_3to1 #(32) bias_mux_3to1_0(.data_in_0(32'b00000000000000000000000000000011), .data_in_1(32'b00000000000000000000000000000011), .data_in_2(32'b00000000000000000000000000000011)
mux_3to1 #(32) bias_mux_3to1_1(.data_in_0(32'b11111111111111111111111111111110), .data_in_1(32'b00000000000000000000000000000000), .data_in_2(32'b00000000000000000000000000000010
mux_3to1 #(32) bias_mux_3to1_2(.data_in_0(32'b00000000000000000000000000000001), .data_in_1(32'b00000000000000000000000000000001), .data_in_2(32'b11111111111111111111111111111111)
mux_3to1 #(32) bias_mux_3to1_3(.data_in_0(32'b00000000000000000000000000000011), .data_in_1(32'b00000000000000000000000000000010), .data_in_2(32'b11111111111111111111111111111111)

//24 MAC modules
MAC MAC_0(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_0), .B(MAC_in_para_0), .out(MAC_out_0));
MAC MAC_1(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_1), .B(MAC_in_para_1), .out(MAC_out_1));
MAC MAC_2(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_2), .B(MAC_in_para_2), .out(MAC_out_2));
MAC MAC_3(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_3), .B(MAC_in_para_3), .out(MAC_out_3));
MAC MAC_4(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_4), .B(MAC_in_para_4), .out(MAC_out_4));
MAC MAC_5(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_5), .B(MAC_in_para_5), .out(MAC_out_5));
MAC MAC_6(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_6), .B(MAC_in_para_6), .out(MAC_out_6));
MAC MAC_7(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_7), .B(MAC_in_para_7), .out(MAC_out_7));
MAC MAC_8(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_8), .B(MAC_in_para_8), .out(MAC_out_8));
MAC MAC_9(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_9), .B(MAC_in_para_9), .out(MAC_out_9));
MAC MAC_10(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_10), .B(MAC_in_para_10), .out(MAC_out_10));
MAC MAC_11(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_11), .B(MAC_in_para_11), .out(MAC_out_11));
MAC MAC_12(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_12), .B(MAC_in_para_12), .out(MAC_out_12));
MAC MAC_13(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_13), .B(MAC_in_para_13), .out(MAC_out_13));
MAC MAC_14(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_14), .B(MAC_in_para_14), .out(MAC_out_14));
MAC MAC_15(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_15), .B(MAC_in_para_15), .out(MAC_out_15));
MAC MAC_16(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_16), .B(MAC_in_para_16), .out(MAC_out_16));
MAC MAC_17(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_17), .B(MAC_in_para_17), .out(MAC_out_17));
MAC MAC_18(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_18), .B(MAC_in_para_18), .out(MAC_out_18));
MAC MAC_19(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_19), .B(MAC_in_para_19), .out(MAC_out_19));
MAC MAC_20(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_20), .B(MAC_in_para_20), .out(MAC_out_20));
MAC MAC_21(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_21), .B(MAC_in_para_21), .out(MAC_out_21));
MAC MAC_22(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_22), .B(MAC_in_para_22), .out(MAC_out_22));
MAC MAC_23(.clk(clk), .enable(MAC_enable), .reset(rMAC), .MAC_layer(MAC_layer), .A(MAC_in_data_23), .B(MAC_in_para_23), .out(MAC_out_23));

//The after MAC does all subsequent operations needed after the MAC depending on the layer. ReLU, combination, biases, shifting, etc.
after_MAC after_MAC_0(.MAC_layer(MAC_layer), .MAC_out_0(MAC_out_0), .MAC_out_1(MAC_out_1), .MAC_out_2(MAC_out_2), .MAC_out_3(MAC_out_3),
                      .MAC_out_4(MAC_out_4), .MAC_out_5(MAC_out_5), .bias_0(32'b11111111111111111111111111110100), .bias_1(32'b11111111111111111111111111110010), .bias_2(32'b11111
                      .bias_3(32'b11111111111111111111111111111100), .bias_4(32'b11111111111111111111111111111100), .bias_5(32'b00000000000000000000000000000100), .conv2_bias(conv
                      .out_1(after_MAC_0_out_1), .out_2(after_MAC_0_out_2), .out_3(after_MAC_0_out_3), .out_4(after_MAC_0_out_4), .out_5(after_MAC_0_out_5), .out_conv2(out_conv2_0
after_MAC after_MAC_1(.MAC_layer(MAC_layer), .MAC_out_0(MAC_out_6), .MAC_out_1(MAC_out_7), .MAC_out_2(MAC_out_8), .MAC_out_3(MAC_out_9),
                      .MAC_out_4(MAC_out_10), .MAC_out_5(MAC_out_11), .bias_0(32'b11111111111111111111111111110100), .bias_1(32'b11111111111111111111111111110010), .bias_2(32'b111
                      .bias_3(32'b11111111111111111111111111111100), .bias_4(32'b11111111111111111111111111111100), .bias_5(32'b00000000000000000000000000000100), .conv2_bias(conv
                      .out_1(after_MAC_1_out_1), .out_2(after_MAC_1_out_2), .out_3(after_MAC_1_out_3), .out_4(conv1_mem_4_0_data_b), .out_5(conv1_mem_5_0_data_b), .out_conv2(out_c
after_MAC after_MAC_2(.MAC_layer(MAC_layer), .MAC_out_0(MAC_out_12), .MAC_out_1(MAC_out_13), .MAC_out_2(MAC_out_14), .MAC_out_3(MAC_out_15),
                      .MAC_out_4(MAC_out_16), .MAC_out_5(MAC_out_17), .bias_0(32'b11111111111111111111111111110100), .bias_1(32'b11111111111111111111111111110010), .bias_2(32'b111
                      .bias_3(32'b11111111111111111111111111111100), .bias_4(32'b11111111111111111111111111111100), .bias_5(32'b00000000000000000000000000000100), .conv2_bias(conv
                      .out_1(conv1_mem_1_1_data_a), .out_2(conv1_mem_2_1_data_a), .out_3(conv1_mem_3_1_data_a), .out_4(conv1_mem_4_1_data_a), .out_5(conv1_mem_5_1_data_a), .out_co
after_MAC after_MAC_3(.MAC_layer(MAC_layer), .MAC_out_0(MAC_out_18), .MAC_out_1(MAC_out_19), .MAC_out_2(MAC_out_20), .MAC_out_3(MAC_out_21),
                      .MAC_out_4(MAC_out_22), .MAC_out_5(MAC_out_23), .bias_0(32'b11111111111111111111111111110100), .bias_1(32'b11111111111111111111111111110010), .bias_2(32'b111
                      .bias_3(32'b11111111111111111111111111111100), .bias_4(32'b11111111111111111111111111111100), .bias_5(32'b00000000000000000000000000000100), .conv2_bias(conv
                      .out_1(conv1_mem_1_1_data_b), .out_2(conv1_mem_2_1_data_b), .out_3(conv1_mem_3_1_data_b), .out_4(conv1_mem_4_1_data_b), .out_5(conv1_mem_5_1_data_b), .out_co

logic signed [15:0] result_0, result_1, result_2, result_3, result_4, result_5, result_6, result_7, result_8, result_9;

demux_1to2 out_conv1_fc_demux_0(.data_in(after_MAC_0_out_0), .sel(FC_layer), .data_out_0(conv1_mem_0_0_data_a), .data_out_1(result_0));
demux_1to2 out_conv1_fc_demux_1(.data_in(after_MAC_0_out_1), .sel(FC_layer), .data_out_0(conv1_mem_1_0_data_a), .data_out_1(result_1));
demux_1to2 out_conv1_fc_demux_2(.data_in(after_MAC_0_out_2), .sel(FC_layer), .data_out_0(conv1_mem_2_0_data_a), .data_out_1(result_2));
demux_1to2 out_conv1_fc_demux_3(.data_in(after_MAC_0_out_3), .sel(FC_layer), .data_out_0(conv1_mem_3_0_data_a), .data_out_1(result_3));
demux_1to2 out_conv1_fc_demux_4(.data_in(after_MAC_0_out_4), .sel(FC_layer), .data_out_0(conv1_mem_4_0_data_a), .data_out_1(result_4));
demux_1to2 out_conv1_fc_demux_5(.data_in(after_MAC_0_out_5), .sel(FC_layer), .data_out_0(conv1_mem_5_0_data_a), .data_out_1(result_5));
demux_1to2 out_conv1_fc_demux_6(.data_in(after_MAC_1_out_0), .sel(FC_layer), .data_out_0(conv1_mem_0_0_data_b), .data_out_1(result_6));
demux_1to2 out_conv1_fc_demux_7(.data_in(after_MAC_1_out_1), .sel(FC_layer), .data_out_0(conv1_mem_1_0_data_b), .data_out_1(result_7));
demux_1to2 out_conv1_fc_demux_8(.data_in(after_MAC_1_out_2), .sel(FC_layer), .data_out_0(conv1_mem_2_0_data_b), .data_out_1(result_8));
demux_1to2 out_conv1_fc_demux_9(.data_in(after_MAC_1_out_3), .sel(FC_layer), .data_out_0(conv1_mem_3_0_data_b), .data_out_1(result_9));

demux_1to3 out_conv2_demux_0(.data_in(out_conv2_0), .sel(conv2_count), .data_out_0(conv2_mem_0_data), .data_out_1(conv2_mem_4_data), .data_out_2(conv2_mem_8_data)); //out img 0, 4
demux_1to3 out_conv2_demux_1(.data_in(out_conv2_1), .sel(conv2_count), .data_out_0(conv2_mem_1_data), .data_out_1(conv2_mem_5_data), .data_out_2(conv2_mem_9_data)); //out img 1, 5
demux_1to3 out_conv2_demux_2(.data_in(out_conv2_2), .sel(conv2_count), .data_out_0(conv2_mem_2_data), .data_out_1(conv2_mem_6_data), .data_out_2(conv2_mem_10_data)); //out img 2,
demux_1to3 out_conv2_demux_3(.data_in(out_conv2_3), .sel(conv2_count), .data_out_0(conv2_mem_3_data), .data_out_1(conv2_mem_7_data), .data_out_2(conv2_mem_11_data)); //out img 3,

//-----------------------------------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------------------------------------------------
//
//  POOLING (DONE)
//
//-----------------------------------------------------------------------------------------------------------------------------------

logic signed [15:0] pooling_0_a0, pooling_0_a1, pooling_0_b0, pooling_0_b1,
                    pooling_1_a0, pooling_1_a1, pooling_1_b0, pooling_1_b1,
                    pooling_2_a0, pooling_2_a1, pooling_2_b0, pooling_2_b1,
                    pooling_3_a0, pooling_3_a1, pooling_3_b0, pooling_3_b1,
                    pooling_4_a0, pooling_4_a1, pooling_4_b0, pooling_4_b1,
                    pooling_5_a0, pooling_5_a1, pooling_5_b0, pooling_5_b1,
                    pooling_6_a0, pooling_6_a1, pooling_6_b0, pooling_6_b1,
                    pooling_7_a0, pooling_7_a1, pooling_7_b0, pooling_7_b1,
                    pooling_8_a0, pooling_8_a1, pooling_8_b0, pooling_8_b1,
                    pooling_9_a0, pooling_9_a1, pooling_9_b0, pooling_9_b1,
                    pooling_10_a0, pooling_10_a1, pooling_10_b0, pooling_10_b1,
                    pooling_11_a0, pooling_11_a1, pooling_11_b0, pooling_11_b1,
                    pooling_0_out, pooling_1_out, pooling_2_out, pooling_3_out,
                    pooling_4_out, pooling_5_out, pooling_6_out, pooling_7_out,
                    pooling_8_out, pooling_9_out, pooling_10_out, pooling_11_out;
mux_2to1 pooling_mux_0_a0(.data_in_0(conv1_mem_0_0_q_a), .data_in_1(conv2_mem_0_0_q_a), .sel(pooling_layer), .data_out(pooling_0_a0));
mux_2to1 pooling_mux_0_a1(.data_in_0(conv1_mem_0_0_q_b), .data_in_1(conv2_mem_0_0_q_b), .sel(pooling_layer), .data_out(pooling_0_a1));
mux_2to1 pooling_mux_0_b0(.data_in_0(conv1_mem_0_1_q_a), .data_in_1(conv2_mem_0_1_q_a), .sel(pooling_layer), .data_out(pooling_0_b0));
mux_2to1 pooling_mux_0_b1(.data_in_0(conv1_mem_0_1_q_b), .data_in_1(conv2_mem_0_1_q_b), .sel(pooling_layer), .data_out(pooling_0_b1));
mux_2to1 pooling_mux_1_a0(.data_in_0(conv1_mem_0_2_q_a), .data_in_1(conv2_mem_1_0_q_a), .sel(pooling_layer), .data_out(pooling_1_a0));
mux_2to1 pooling_mux_1_a1(.data_in_0(conv1_mem_0_2_q_b), .data_in_1(conv2_mem_1_0_q_b), .sel(pooling_layer), .data_out(pooling_1_a1));

//mux_2to1 pooling_mux_0_b0(.data_in_0(conv1_mem_0_2_q_a), .data_in_1(conv2_mem_0_1_q_a), .sel(pooling_layer), .data_out(pooling_0_b0));
//mux_2to1 pooling_mux_0_b1(.data_in_0(conv1_mem_0_2_q_b), .data_in_1(conv2_mem_0_1_q_b), .sel(pooling_layer), .data_out(pooling_0_b1));
//mux_2to1 pooling_mux_1_a0(.data_in_0(conv1_mem_0_1_q_a), .data_in_1(conv2_mem_1_0_q_a), .sel(pooling_layer), .data_out(pooling_1_a0));
//mux_2to1 pooling_mux_1_a1(.data_in_0(conv1_mem_0_1_q_b), .data_in_1(conv2_mem_1_0_q_b), .sel(pooling_layer), .data_out(pooling_1_a1));
mux_2to1 pooling_mux_1_b0(.data_in_0(conv1_mem_0_3_q_a), .data_in_1(conv2_mem_1_1_q_a), .sel(pooling_layer), .data_out(pooling_1_b0));
mux_2to1 pooling_mux_1_b1(.data_in_0(conv1_mem_0_3_q_b), .data_in_1(conv2_mem_1_1_q_b), .sel(pooling_layer), .data_out(pooling_1_b1));
mux_2to1 pooling_mux_2_a0(.data_in_0(conv1_mem_1_0_q_a), .data_in_1(conv2_mem_2_0_q_a), .sel(pooling_layer), .data_out(pooling_2_a0));
mux_2to1 pooling_mux_2_a1(.data_in_0(conv1_mem_1_0_q_b), .data_in_1(conv2_mem_2_0_q_b), .sel(pooling_layer), .data_out(pooling_2_a1));
mux_2to1 pooling_mux_2_b0(.data_in_0(conv1_mem_1_1_q_a), .data_in_1(conv2_mem_2_1_q_a), .sel(pooling_layer), .data_out(pooling_2_b0));
mux_2to1 pooling_mux_2_b1(.data_in_0(conv1_mem_1_1_q_b), .data_in_1(conv2_mem_2_1_q_b), .sel(pooling_layer), .data_out(pooling_2_b1));
mux_2to1 pooling_mux_3_a0(.data_in_0(conv1_mem_1_2_q_a), .data_in_1(conv2_mem_3_0_q_a), .sel(pooling_layer), .data_out(pooling_3_a0));
mux_2to1 pooling_mux_3_a1(.data_in_0(conv1_mem_1_2_q_b), .data_in_1(conv2_mem_3_0_q_b), .sel(pooling_layer), .data_out(pooling_3_a1));
mux_2to1 pooling_mux_3_b0(.data_in_0(conv1_mem_1_3_q_a), .data_in_1(conv2_mem_3_1_q_a), .sel(pooling_layer), .data_out(pooling_3_b0));
mux_2to1 pooling_mux_3_b1(.data_in_0(conv1_mem_1_3_q_b), .data_in_1(conv2_mem_3_1_q_b), .sel(pooling_layer), .data_out(pooling_3_b1));
mux_2to1 pooling_mux_4_a0(.data_in_0(conv1_mem_2_0_q_a), .data_in_1(conv2_mem_4_0_q_a), .sel(pooling_layer), .data_out(pooling_4_a0));
mux_2to1 pooling_mux_4_a1(.data_in_0(conv1_mem_2_0_q_b), .data_in_1(conv2_mem_4_0_q_b), .sel(pooling_layer), .data_out(pooling_4_a1));
mux_2to1 pooling_mux_4_b0(.data_in_0(conv1_mem_2_1_q_a), .data_in_1(conv2_mem_4_1_q_a), .sel(pooling_layer), .data_out(pooling_4_b0));
mux_2to1 pooling_mux_4_b1(.data_in_0(conv1_mem_2_1_q_b), .data_in_1(conv2_mem_4_1_q_b), .sel(pooling_layer), .data_out(pooling_4_b1));
mux_2to1 pooling_mux_5_a0(.data_in_0(conv1_mem_2_2_q_a), .data_in_1(conv2_mem_5_0_q_a), .sel(pooling_layer), .data_out(pooling_5_a0));
mux_2to1 pooling_mux_5_a1(.data_in_0(conv1_mem_2_2_q_b), .data_in_1(conv2_mem_5_0_q_b), .sel(pooling_layer), .data_out(pooling_5_a1));
mux_2to1 pooling_mux_5_b0(.data_in_0(conv1_mem_2_3_q_a), .data_in_1(conv2_mem_5_1_q_a), .sel(pooling_layer), .data_out(pooling_5_b0));
mux_2to1 pooling_mux_5_b1(.data_in_0(conv1_mem_2_3_q_b), .data_in_1(conv2_mem_5_1_q_b), .sel(pooling_layer), .data_out(pooling_5_b1));
mux_2to1 pooling_mux_6_a0(.data_in_0(conv1_mem_3_0_q_a), .data_in_1(conv2_mem_6_0_q_a), .sel(pooling_layer), .data_out(pooling_6_a0));
mux_2to1 pooling_mux_6_a1(.data_in_0(conv1_mem_3_0_q_b), .data_in_1(conv2_mem_6_0_q_b), .sel(pooling_layer), .data_out(pooling_6_a1));
mux_2to1 pooling_mux_6_b0(.data_in_0(conv1_mem_3_1_q_a), .data_in_1(conv2_mem_6_1_q_a), .sel(pooling_layer), .data_out(pooling_6_b0));
mux_2to1 pooling_mux_6_b1(.data_in_0(conv1_mem_3_1_q_b), .data_in_1(conv2_mem_6_1_q_b), .sel(pooling_layer), .data_out(pooling_6_b1));
mux_2to1 pooling_mux_7_a0(.data_in_0(conv1_mem_3_2_q_a), .data_in_1(conv2_mem_7_0_q_a), .sel(pooling_layer), .data_out(pooling_7_a0));
```

```
mux_2to1 pooling_mux_7_a1(.data_in_0(conv1_mem_3_2_q_b), .data_in_1(conv2_mem_7_0_q_b), .sel(pooling_layer), .data_out(pooling_7_a1));
mux_2to1 pooling_mux_7_b0(.data_in_0(conv1_mem_3_3_q_a), .data_in_1(conv2_mem_7_1_q_a), .sel(pooling_layer), .data_out(pooling_7_b0));
mux_2to1 pooling_mux_7_b1(.data_in_0(conv1_mem_3_3_q_b), .data_in_1(conv2_mem_7_1_q_b), .sel(pooling_layer), .data_out(pooling_7_b1));
mux_2to1 pooling_mux_8_a0(.data_in_0(conv1_mem_4_0_q_a), .data_in_1(conv2_mem_8_0_q_a), .sel(pooling_layer), .data_out(pooling_8_a0));
mux_2to1 pooling_mux_8_a1(.data_in_0(conv1_mem_4_0_q_b), .data_in_1(conv2_mem_8_0_q_b), .sel(pooling_layer), .data_out(pooling_8_a1));
mux_2to1 pooling_mux_8_b0(.data_in_0(conv1_mem_4_1_q_a), .data_in_1(conv2_mem_8_1_q_a), .sel(pooling_layer), .data_out(pooling_8_b0));
mux_2to1 pooling_mux_8_b1(.data_in_0(conv1_mem_4_1_q_b), .data_in_1(conv2_mem_8_1_q_b), .sel(pooling_layer), .data_out(pooling_8_b1));
mux_2to1 pooling_mux_9_a0(.data_in_0(conv1_mem_4_2_q_a), .data_in_1(conv2_mem_9_0_q_a), .sel(pooling_layer), .data_out(pooling_9_a0));
mux_2to1 pooling_mux_9_a1(.data_in_0(conv1_mem_4_2_q_b), .data_in_1(conv2_mem_9_0_q_b), .sel(pooling_layer), .data_out(pooling_9_a1));
mux_2to1 pooling_mux_9_b0(.data_in_0(conv1_mem_4_3_q_a), .data_in_1(conv2_mem_9_1_q_a), .sel(pooling_layer), .data_out(pooling_9_b0));
mux_2to1 pooling_mux_9_b1(.data_in_0(conv1_mem_4_3_q_b), .data_in_1(conv2_mem_9_1_q_b), .sel(pooling_layer), .data_out(pooling_9_b1));
mux_2to1 pooling_mux_10_a0(.data_in_0(conv1_mem_5_0_q_a), .data_in_1(conv2_mem_10_0_q_a), .sel(pooling_layer), .data_out(pooling_10_a0));
mux_2to1 pooling_mux_10_a1(.data_in_0(conv1_mem_5_0_q_b), .data_in_1(conv2_mem_10_0_q_b), .sel(pooling_layer), .data_out(pooling_10_a1));
mux_2to1 pooling_mux_10_b0(.data_in_0(conv1_mem_5_1_q_a), .data_in_1(conv2_mem_10_1_q_a), .sel(pooling_layer), .data_out(pooling_10_b0));
mux_2to1 pooling_mux_10_b1(.data_in_0(conv1_mem_5_1_q_b), .data_in_1(conv2_mem_10_1_q_b), .sel(pooling_layer), .data_out(pooling_10_b1));
mux_2to1 pooling_mux_11_a0(.data_in_0(conv1_mem_5_2_q_a), .data_in_1(conv2_mem_11_0_q_a), .sel(pooling_layer), .data_out(pooling_11_a0));
mux_2to1 pooling_mux_11_a1(.data_in_0(conv1_mem_5_2_q_b), .data_in_1(conv2_mem_11_0_q_b), .sel(pooling_layer), .data_out(pooling_11_a1));
mux_2to1 pooling_mux_11_b0(.data_in_0(conv1_mem_5_3_q_a), .data_in_1(conv2_mem_11_1_q_a), .sel(pooling_layer), .data_out(pooling_11_b0));
mux_2to1 pooling_mux_11_b1(.data_in_0(conv1_mem_5_3_q_b), .data_in_1(conv2_mem_11_1_q_b), .sel(pooling_layer), .data_out(pooling_11_b1));

//12 pooling modules to be used by pooling layers
pooling pooling_0(.a_0(pooling_0_a0), .a_1(pooling_0_a1), .b_0(pooling_1_a0), .b_1(pooling_1_a1), .out(pooling_0_out));
pooling pooling_1(.a_0(pooling_0_b0), .a_1(pooling_0_b1), .b_0(pooling_1_b0), .b_1(pooling_1_b1), .out(pooling_1_out));
pooling pooling_2(.a_0(pooling_2_a0), .a_1(pooling_2_a1), .b_0(pooling_3_a0), .b_1(pooling_3_a1), .out(pooling_2_out));
pooling pooling_3(.a_0(pooling_2_b0), .a_1(pooling_2_b1), .b_0(pooling_3_b0), .b_1(pooling_3_b1), .out(pooling_3_out));
pooling pooling_4(.a_0(pooling_4_a0), .a_1(pooling_4_a1), .b_0(pooling_5_a0), .b_1(pooling_5_a1), .out(pooling_4_out));
pooling pooling_5(.a_0(pooling_4_b0), .a_1(pooling_4_b1), .b_0(pooling_5_b0), .b_1(pooling_5_b1), .out(pooling_5_out));
pooling pooling_6(.a_0(pooling_6_a0), .a_1(pooling_6_a1), .b_0(pooling_7_a0), .b_1(pooling_7_a1), .out(pooling_6_out));
pooling pooling_7(.a_0(pooling_6_b0), .a_1(pooling_6_b1), .b_0(pooling_7_b0), .b_1(pooling_7_b1), .out(pooling_7_out));
pooling pooling_8(.a_0(pooling_8_a0), .a_1(pooling_8_a1), .b_0(pooling_9_a0), .b_1(pooling_9_a1), .out(pooling_8_out));
pooling pooling_9(.a_0(pooling_8_b0), .a_1(pooling_8_b1), .b_0(pooling_9_b0), .b_1(pooling_9_b1), .out(pooling_9_out));
pooling pooling_10(.a_0(pooling_10_a0), .a_1(pooling_10_a1), .b_0(pooling_11_a0), .b_1(pooling_11_a1), .out(pooling_10_out));
pooling pooling_11(.a_0(pooling_10_b0), .a_1(pooling_10_b1), .b_0(pooling_11_b0), .b_1(pooling_11_b1), .out(pooling_11_out));

demux_1to2 pooling_demux_out_0(.data_in(pooling_0_out), .sel(pooling_layer), .data_out_0(p1_mem_0_data_a), .data_out_1(P2_mem_0_data));
demux_1to2 pooling_demux_out_1(.data_in(pooling_1_out), .sel(pooling_layer), .data_out_0(p1_mem_0_data_b), .data_out_1(P2_mem_1_data));
demux_1to2 pooling_demux_out_2(.data_in(pooling_2_out), .sel(pooling_layer), .data_out_0(p1_mem_1_data_a), .data_out_1(P2_mem_2_data));
demux_1to2 pooling_demux_out_3(.data_in(pooling_3_out), .sel(pooling_layer), .data_out_0(p1_mem_1_data_b), .data_out_1(P2_mem_3_data));
demux_1to2 pooling_demux_out_4(.data_in(pooling_4_out), .sel(pooling_layer), .data_out_0(p1_mem_2_data_a), .data_out_1(P2_mem_4_data));
demux_1to2 pooling_demux_out_5(.data_in(pooling_5_out), .sel(pooling_layer), .data_out_0(p1_mem_2_data_b), .data_out_1(P2_mem_5_data));
demux_1to2 pooling_demux_out_6(.data_in(pooling_6_out), .sel(pooling_layer), .data_out_0(p1_mem_3_data_a), .data_out_1(P2_mem_6_data));
demux_1to2 pooling_demux_out_7(.data_in(pooling_7_out), .sel(pooling_layer), .data_out_0(p1_mem_3_data_b), .data_out_1(P2_mem_7_data));
demux_1to2 pooling_demux_out_8(.data_in(pooling_8_out), .sel(pooling_layer), .data_out_0(p1_mem_4_data_a), .data_out_1(P2_mem_8_data));
demux_1to2 pooling_demux_out_9(.data_in(pooling_9_out), .sel(pooling_layer), .data_out_0(p1_mem_4_data_b), .data_out_1(P2_mem_9_data));
demux_1to2 pooling_demux_out_10(.data_in(pooling_10_out), .sel(pooling_layer), .data_out_0(p1_mem_5_data_a), .data_out_1(P2_mem_10_data));
demux_1to2 pooling_demux_out_11(.data_in(pooling_11_out), .sel(pooling_layer), .data_out_0(p1_mem_5_data_b), .data_out_1(P2_mem_11_data));

//----------------------------------------------------------------------------------------------------------------------------------------------
//----------------------------------------------------------------------------------------------------------------------------------------------
//----------------------------------------------------------------------------------------------------------------------------------------------

endmodule


module CNN_ctrl(input logic reset, img_mem_read_done, conv1_mem_write_done, conv1_mem_read_done, conv1_k_mem_read_done, P1_mem_read_done, P1_mem_write_done,
                            conv2_mem_write_done, conv2_mem_read_done, conv2_k_mem_read_done, P2_mem_write_done, P2_mem_read_done, fc_mem_read_done,
                input logic [7:0] ctrl,
                output logic pooling_layer, rMAC, MAC_enable, Conv1_layer, P1_layer, Conv2_layer, P2_layer, FC_layer,
                             img_mem_read_reset, conv1_mem_write_reset, conv1_mem_read_reset, conv1_k_mem_read_reset, P1_mem_read_reset, P1_mem_write_reset,
                             conv2_mem_write_reset, conv2_mem_read_reset, conv2_k_mem_read_reset, P2_mem_write_reset, P2_mem_read_reset, fc_mem_read_reset, img_load,
                output logic [7:0] return_ctrl,
                output logic [1:0] MAC_layer);

    always_comb begin
        if (reset == 1'b1 || ctrl == 8'b00000000) begin
            return_ctrl = 8'b00000000;
            img_load = 1'b0;
            MAC_layer = 2'b00;
            pooling_layer = 1'b0;
            rMAC = 1'b1;
            MAC_enable = 1'b0;
            Conv1_layer = 1'b0;
            P1_layer = 1'b0;
            Conv2_layer = 1'b0;
            P2_layer = 1'b0;
            FC_layer = 1'b0;
            img_mem_read_reset = 1'b1;
            conv1_mem_write_reset = 1'b1;
            conv1_mem_read_reset = 1'b1;
            conv1_k_mem_read_reset = 1'b1;
            P1_mem_read_reset = 1'b1;
            P1_mem_write_reset = 1'b1;
            conv2_mem_write_reset = 1'b1;
            conv2_mem_read_reset = 1'b1;
            conv2_k_mem_read_reset = 1'b1;
            P2_mem_write_reset = 1'b1;
            P2_mem_read_reset = 1'b1;
            fc_mem_read_reset = 1'b1;
        end

        else if(ctrl == 8'b00000001) begin //img load
            MAC_layer = 2'b00;
            img_load = 1'b1;
            pooling_layer = 1'b0;
            rMAC = 1'b1;
            MAC_enable = 1'b0;
            Conv1_layer = 1'b0;
            P1_layer = 1'b0;
            Conv2_layer = 1'b0;
            P2_layer = 1'b0;
            FC_layer = 1'b0;
            img_mem_read_reset = 1'b1;
            conv1_mem_write_reset = 1'b1;
            conv1_mem_read_reset = 1'b1;
            conv1_k_mem_read_reset = 1'b1;
            P1_mem_read_reset = 1'b1;
            P1_mem_write_reset = 1'b1;
            conv2_mem_write_reset = 1'b1;
            conv2_mem_read_reset = 1'b1;
            conv2_k_mem_read_reset = 1'b1;
            P2_mem_write_reset = 1'b1;
            P2_mem_read_reset = 1'b1;
            fc_mem_read_reset = 1'b1;
            //TODO: Need mem read done
            //if(img_mem_read_done == 1'b1) begin
            //    return_ctrl = 8'b00000001;
            //end
            //else return_ctrl = 0;
            return_ctrl = 0;
        end
        else if(ctrl == 8'b00000010) begin //conv1
            img_load = 1'b0;
            MAC_layer = 2'b00;
            pooling_layer = 1'b0;
            rMAC = 1'b0;
            MAC_enable = 1'b1;
            Conv1_layer = 1'b1;
            P1_layer = 1'b0;
            Conv2_layer = 1'b0;
            P2_layer = 1'b0;
            FC_layer = 1'b0;
            img_mem_read_reset = 1'b0;
            conv1_mem_write_reset = 1'b0;
            conv1_mem_read_reset = 1'b1;
            conv1_k_mem_read_reset = 1'b0;
            P1_mem_read_reset = 1'b1;
            P1_mem_write_reset = 1'b1;
            conv2_mem_write_reset = 1'b1;
            conv2_mem_read_reset = 1'b1;
            conv2_k_mem_read_reset = 1'b1;
            P2_mem_write_reset = 1'b1;
            P2_mem_read_reset = 1'b1;
            fc_mem_read_reset = 1'b1;
            if(img_mem_read_done == 1'b1 && conv1_mem_write_done == 1'b1 && conv1_k_mem_read_done == 1'b1) begin
                return_ctrl = 8'b00000010;
            end
            else return_ctrl = 8'b00000001;
        end
        else if(ctrl == 8'b00000011) begin //pool1
            img_load = 1'b0;
            MAC_layer = 2'b00;
            pooling_layer = 1'b0;
            rMAC = 1'b1;
            MAC_enable = 1'b0;
            Conv1_layer = 1'b0;
            P1_layer = 1'b1;
            Conv2_layer = 1'b0;
            P2_layer = 1'b0;
            FC_layer = 1'b0;
            img_mem_read_reset = 1'b1;
            conv1_mem_write_reset = 1'b1;
            conv1_mem_read_reset = 1'b0;
            conv1_k_mem_read_reset = 1'b1;
            P1_mem_read_reset = 1'b1;
            P1_mem_write_reset = 1'b0;
            conv2_mem_write_reset = 1'b1;
            conv2_mem_read_reset = 1'b1;
            conv2_k_mem_read_reset = 1'b1;
            P2_mem_write_reset = 1'b1;
            P2_mem_read_reset = 1'b1;
            fc_mem_read_reset = 1'b1;
            if(conv1_mem_read_done == 1'b1 && P1_mem_write_done == 1'b1) begin
                return_ctrl = 8'b00000011;
            end
            else return_ctrl = 8'b00000010;
        end
        else if(ctrl == 8'b00000100) begin //conv2
            img_load = 1'b0;
            MAC_layer = 2'b01;
            pooling_layer = 1'b0;
            rMAC = 1'b0;
            MAC_enable = 1'b1;
            Conv1_layer = 1'b0;
            P1_layer = 1'b0;
            Conv2_layer = 1'b1;
            P2_layer = 1'b0;
            FC_layer = 1'b0;
            img_mem_read_reset = 1'b1;
            conv1_mem_write_reset = 1'b1;
            conv1_mem_read_reset = 1'b1;
            conv1_k_mem_read_reset = 1'b1;
            P1_mem_read_reset = 1'b0;
            P1_mem_write_reset = 1'b1;
            conv2_mem_write_reset = 1'b0;
            conv2_mem_read_reset = 1'b1;
            conv2_k_mem_read_reset = 1'b0;
            P2_mem_write_reset = 1'b1;
            P2_mem_read_reset = 1'b1;
            fc_mem_read_reset = 1'b1;
            if(P1_mem_read_done  == 1'b1 && conv2_mem_write_done == 1'b1 && conv2_k_mem_read_done == 1'b1) begin
                return_ctrl = 8'b00000100;
            end
            else return_ctrl = 8'b00000011;
        end
        else if(ctrl == 8'b00000101) begin //pool2
            img_load = 1'b0;
            MAC_layer = 2'b00;
            pooling_layer = 1'b1;
            rMAC = 1'b1;
            MAC_enable = 1'b0;
            Conv1_layer = 1'b0;
            P1_layer = 1'b0;
            Conv2_layer = 1'b0;
            P2_layer = 1'b1;
            FC_layer = 1'b0;
            img_mem_read_reset = 1'b1;
            conv1_mem_write_reset = 1'b1;
            conv1_mem_read_reset = 1'b1;
            conv1_k_mem_read_reset = 1'b1;
            P1_mem_read_reset = 1'b1;
            P1_mem_write_reset = 1'b1;
            conv2_mem_write_reset = 1'b1;
            conv2_mem_read_reset = 1'b0;
            conv2_k_mem_read_reset = 1'b1;
            P2_mem_write_reset = 1'b0;
            P2_mem_read_reset = 1'b1;
            fc_mem_read_reset = 1'b1;
            if(conv2_mem_read_done == 1'b1 && P2_mem_write_done == 1'b1) begin
                return_ctrl = 8'b00000101;
            end
            else return_ctrl = 8'b00000100;
        end
        else if(ctrl == 8'b00000110) begin //FC
            img_load = 1'b0;
            MAC_layer = 2'b10;
            pooling_layer = 1'b0;
            rMAC = 1'b0;
            MAC_enable = 1'b1;
            Conv1_layer = 1'b0;
            P1_layer = 1'b0;
            Conv2_layer = 1'b0;
            P2_layer = 1'b0;
            FC_layer = 1'b1;
            img_mem_read_reset = 1'b1;
            conv1_mem_write_reset = 1'b1;
            conv1_mem_read_reset = 1'b1;
            conv1_k_mem_read_reset = 1'b1;
            P1_mem_read_reset = 1'b1;
            P1_mem_write_reset = 1'b1;
            conv2_mem_write_reset = 1'b1;
            conv2_mem_read_reset = 1'b1;
            conv2_k_mem_read_reset = 1'b1;
            P2_mem_write_reset = 1'b1;
            P2_mem_read_reset = 1'b0;
            fc_mem_read_reset = 1'b0;
            if(P2_mem_read_done == 1'b1 && fc_mem_read_done == 1'b1) begin
                return_ctrl = 8'b00000110;
            end
            else return_ctrl = 8'b00000101;
        end
        else begin
            return_ctrl = 8'b00000000;
            img_load = 1'b0;
            MAC_layer = 2'b00;
            pooling_layer = 1'b0;
            rMAC = 1'b1;
            MAC_enable = 1'b0;
            Conv1_layer = 1'b0;
            P1_layer = 1'b0;
            Conv2_layer = 1'b0;
            P2_layer = 1'b0;
            FC_layer = 1'b0;
            img_mem_read_reset = 1'b1;
            conv1_mem_write_reset = 1'b1;
            conv1_mem_read_reset = 1'b1;
            conv1_k_mem_read_reset = 1'b1;
            P1_mem_read_reset = 1'b1;
            P1_mem_write_reset = 1'b1;
```

```
            conv2_mem_write_reset = 1'b1;
            conv2_mem_read_reset = 1'b1;
            conv2_k_mem_read_reset = 1'b1;
            P2_mem_write_reset = 1'b1;
            P2_mem_read_reset = 1'b1;
            fc_mem_read_reset = 1'b1;
        end
    end

endmodule


#include <iostream>
#include "VCNN.h"
#include <verilated.h>
#include <verilated_vcd_c.h>
#define IMAGE_METADATA_OFFSET 16

int gtime;
double sc_time_stamp () {       // Called by $time in Verilog
        return gtime;
}

int ctrl = 0;
void increment_control(VCNN* dut) {
  dut->address = 0;
  dut->write = 1;
  dut->read = 0;
  dut->writedata = ctrl + 1;
  ctrl++;
}

void write_img_address(VCNN* dut, int addr) {
  dut->address = 2;
  dut->write = 1;
  dut->read = 0;
  dut->writedata = addr;
}

void write_img_data(VCNN* dut, signed short data) {
  dut->address = 3;
  dut->write = 1;
  dut->read = 0;
  memcpy(&dut->writedata, &data, sizeof(signed short));
}

void get_return_ctrl(VCNN* dut) {
  dut->address = 1;
  dut->write = 0;
  dut->read = 1;
  dut->writedata = 0;
}

void dummy_op(VCNN* dut) {
  dut->address = 0;
  dut->write = 0;
  dut->read = 0;
  dut->writedata = 0;
}

int main(int argc, const char ** argv, const char ** env) {
  Verilated::commandArgs(argc, argv);
  //Verilated::debug(1);
  gtime = 0;

  unsigned char raw_image[28 * 28];
  signed short image_data[28 * 28];
  FILE* fp = fopen("../mnist/t10k-images-idx3-ubyte", "rb");
  if (!fp) {
    fprintf(stderr, "Could not read file\n");
    exit(1);
  }

  //dummy read, needed because of file format
  fread(raw_image, 1, IMAGE_METADATA_OFFSET, fp);
  fread(raw_image, 1, 28 * 28, fp);
  fclose(fp);


  for (int i = 0; i < 784; i++) {
    image_data[i] = raw_image[i] * 16;
    if (image_data[i] > 0) {
      //fprintf(stderr, "image_data[%d] = %d\n", i, image_data[i]);
    }
  }
  /*for (int i = 0; i < 784; i++) {
    image_data[i] = i;
  }*/
  // Treat the argument on the command-line as the place to start
  //int n;
  //if (argc > 1 && argv[1][0] != '+') n = atoi(argv[1]);
  //else n = 7; // Default

  VCNN * dut = new VCNN;  // Instantiate the collatz module

  // Enable dumping a VCD file

  Verilated::traceEverOn(true);
  VerilatedVcdC * tfp = new VerilatedVcdC;
  dut->trace(tfp, 99);
  tfp->open("CNN.vcd");

  // Initial values

  dut->clk = 0;
  dut->reset = 1;
  dut->write = 0;
  dut->read = 0;
  dut->chipselect = 1;
  dut->writedata = 0;
  dut->address = 0;

  //std::cout << dut->n; // Print the starting value of the sequence

  unsigned image_pos = 0;
  unsigned set_addr = 0;
  bool last_clk = true;
  for (gtime = 0 ; gtime < 400000 ; gtime += 10) {
    dut->clk = ((gtime % 20) >= 10) ? 1 : 0; // Simulate a 50 MHz clock
    if (gtime == 20) dut->reset = 1; // Pulse "reset" for two cycles
    if (gtime == 60) dut->reset = 0;
    if (gtime == 80) {
        increment_control(dut);
    }
    else if (gtime > 80 && image_pos < 28 * 28) {
        if (set_addr == 0 && gtime % 20 == 0) {
            write_img_address(dut, image_pos);
            set_addr = 1;
        }
        else if (gtime % 20 == 0) {
            write_img_data(dut, image_data[image_pos]);
            image_pos++;
            set_addr = 0;
        }
    }
    else if (image_pos == 28 * 28 && ctrl == 1) {
      increment_control(dut);
    }
    else if (image_pos >= 28 * 28 && gtime % 20 == 0) {
      if (dut->readdata == ctrl && ctrl < 6) {
        increment_control(dut);
      }
      else if (dut->readdata == ctrl && ctrl == 6) {
        //TODO: we are done, read output
        dummy_op(dut);
      }
      else {
        get_return_ctrl(dut);
      }
    }

    dut->eval();     // Run the simulation for a cycle
    tfp->dump(gtime); // Write the VCD file for this cycle

    last_clk = dut->clk;
  }

  tfp->close(); // Stop dumping the VCD file
  delete tfp;

  dut->final(); // Stop the simulation
  delete dut;

  return 0;
}


module MAC (input logic clk, enable, reset,
                        input logic [1:0] MAC_layer,
                        input logic signed [15:0] A, B,
                        output logic signed [31:0] out);

        reg signed [31:0] MAC_out;
        reg [7:0] count;

        always_ff @(posedge clk or posedge reset) begin
                if(reset == 1'b1) begin
                        MAC_out <= 32'b00000000000000000000000000000000;
                        out <= 32'b00000000000000000000000000000000;
                        count <= 8'b00000000;
                end
                else if (enable == 1'b1) begin
                        if((count == 8'b00011000 && (MAC_layer == 2'b00 || MAC_layer == 2'b01)) || (count == 8'b11000000 && MAC_layer == 2'b10)) begin //if finished with a conv(co
                                out <= MAC_out;
                                MAC_out <= 32'b00000000000000000000000000000000;
                                count <= 8'b00000000;
                        end
                        else begin
                                MAC_out <= MAC_out + (A * B); //does the MAC thing
                                count <= count + 1'b1;
                        end
                end
        end
endmodule


module after_MAC (input logic [1:0] MAC_layer,
                                  input logic signed [31:0] MAC_out_0, MAC_out_1, MAC_out_2, MAC_out_3, MAC_out_4, MAC_out_5, bias_0, bias_1, bias_2, bias_3, bias_4, bias_5, conv2
                                  output logic signed [15:0] out_0, out_1, out_2, out_3, out_4, out_5, out_conv2);

        //bias adding and adding of itermediates for conv1 and conv2
        wire signed [31:0] conv1_interm_0;
        wire signed [31:0] conv1_interm_1;
        wire signed [31:0] conv1_interm_2;
        wire signed [31:0] conv1_interm_3;
        wire signed [31:0] conv1_interm_4;
        wire signed [31:0] conv1_interm_5;
        wire signed [31:0] conv2_interm;
        assign conv2_interm = MAC_out_0 + MAC_out_1 + MAC_out_2 + MAC_out_3 + MAC_out_4 + MAC_out_5 + conv2_bias;
        assign conv1_interm_0 = MAC_out_0 + -32'd188;
        assign conv1_interm_1 = MAC_out_1 + -32'd224;
        assign conv1_interm_2 = MAC_out_2 + -32'd144;
        assign conv1_interm_3 = MAC_out_3 + -32'd64;
        assign conv1_interm_4 = MAC_out_4 + -32'd64;
        assign conv1_interm_5 = MAC_out_5 +  32'd64;

        always_comb begin

                if(MAC_layer == 2'b00) begin //if conv1 performs ReLU and shifts intermediate
                        if(conv1_interm_0[31] == 1'b1) begin
                                out_0[15:0] = 16'b0000000000000000;
                        end
                        else begin
                                out_0[15:0] = conv1_interm_0[19:4];
                        end
                        if(conv1_interm_1[31] == 1'b1) begin
                                out_1[15:0] = 16'b0000000000000000;
                        end
                        else begin
                                out_1[15:0] = conv1_interm_1[19:4];
                        end
                        if(conv1_interm_2[31] == 1'b1) begin
                                out_2[15:0] = 16'b0000000000000000;
                        end
                        else begin
                                out_2[15:0] = conv1_interm_2[19:4];
                        end
                        if(conv1_interm_3[31] == 1'b1) begin
                                out_3[15:0] = 16'b0000000000000000;
                        end
                        else begin
                                out_3[15:0] = conv1_interm_3[19:4];
                        end
                        if(conv1_interm_4[31] == 1'b1) begin
                                out_4[15:0] = 16'b0000000000000000;
                        end
                        else begin
                                out_4[15:0] = conv1_interm_4[19:4];
                        end
                        if(conv1_interm_5[31] == 1'b1) begin
                                out_5[15:0] = 16'b0000000000000000;
                        end
                        else begin
                                out_5[15:0] = conv1_interm_5[19:4];
                        end
            out_conv2 = 16'b0000000000000000;
                end

                else if(MAC_layer == 2'b01) begin //if conv2 performs ReLU and shifts intermediate
                        out_0 = 16'b0000000000000000;
            out_1 = 16'b0000000000000000;
            out_2 = 16'b0000000000000000;
            out_3 = 16'b0000000000000000;
            out_4 = 16'b0000000000000000;
            out_5 = 16'b0000000000000000;
                        if(conv2_interm[31] == 1'b1) begin
                                out_conv2[15:0] = 16'b0000000000000000;
                        end
                        else begin
                                out_conv2[15:0] = conv2_interm[19:4];
                        end
                end

                else if(MAC_layer == 2'b10) begin //if Fully connected layer just shifts
                        out_0[15:0] = MAC_out_0[19:4];
                        out_1[15:0] = MAC_out_1[19:4];
                        out_2[15:0] = MAC_out_2[19:4];
                        out_3[15:0] = MAC_out_3[19:4];
                        out_4[15:0] = MAC_out_4[19:4];
                        out_5[15:0] = MAC_out_5[19:4];
            out_conv2 = 16'b0000000000000000;
                end
        else begin
                        out_0 = 16'b0000000000000000;
                        out_1 = 16'b0000000000000000;
                        out_2 = 16'b0000000000000000;
                        out_3 = 16'b0000000000000000;
                        out_4 = 16'b0000000000000000;
                        out_5 = 16'b0000000000000000;
            out_conv2 = 16'b0000000000000000;
        end
        end
endmodule


module pooling(input logic signed [15:0] a_0, a_1, b_0, b_1,
              output logic signed [15:0] out);

    always_comb begin
        out = (a_0 + a_1 + b_0 + b_1) >> 2;
    end
endmodule


module demux_1to2 #(parameter WORD_SIZE = 16)
                   (input logic [WORD_SIZE - 1:0] data_in,
                    input logic sel,
                    output logic [WORD_SIZE - 1:0] data_out_0, data_out_1);

    always_comb begin
        case (sel)
            1'd0: begin
                data_out_0 = data_in;
                data_out_1 = 16'b0000000000000000;
            end
            1'd1: begin
                data_out_1 = data_in;
                data_out_0 = 16'b0000000000000000;
            end
        endcase
    end
endmodule


module mux_12to1 (input logic [15:0] data_in_0, data_in_1, data_in_2, data_in_3, data_in_4, data_in_5,
                                     data_in_6, data_in_7, data_in_8, data_in_9, data_in_10, data_in_11,
                  input logic [3:0] sel,
                  output logic [15:0] data_out);

    always_comb begin
        case (sel)
            4'b0000: data_out = data_in_0;
            4'b0001: data_out = data_in_1;
            4'b0010: data_out = data_in_2;
            4'b0011: data_out = data_in_3;
            4'b0100: data_out = data_in_4;
            4'b0101: data_out = data_in_5;
            4'b0110: data_out = data_in_6;
            4'b0111: data_out = data_in_7;
            4'b1000: data_out = data_in_8;
            4'b1001: data_out = data_in_9;
            4'b1010: data_out = data_in_10;
            4'b1011: data_out = data_in_11;
            default: data_out = 16'b0000000000000000;
        endcase
    end
endmodule


module mux_2to1 #(parameter WORD_SIZE = 16)
                 (input logic [WORD_SIZE - 1:0] data_in_0, data_in_1,
                  input logic sel,
                  output logic [WORD_SIZE - 1:0] data_out);
    always_comb begin
        case (sel)
            1'd0: data_out = data_in_0;
            1'd1: data_out = data_in_1;
        endcase
    end
endmodule


module mux_3to1 #(parameter WORD_SIZE = 16)
                 (input logic [WORD_SIZE - 1 : 0] data_in_0, data_in_1, data_in_2,
                  input logic [1:0] sel,
                  output logic [WORD_SIZE - 1 : 0] data_out);

    always_comb begin
        case (sel)
            2'b00: data_out = data_in_0;
            2'b01: data_out = data_in_1;
            2'b10: data_out = data_in_2;
            default: data_out = 0;
        endcase
    end
endmodule


module conv1_k_g0_mem ( input [5:0] address_a, address_b,
                                            input clock,
                                            output reg [15:0] q_a, q_b);

        // Declare the ROM variable
        reg [15:0] rom[63:0];

        // Initialize the ROM with $readmemb.  Put the memory contents
        // in the file dual_port_rom_init.txt.  Without this file,
        // this design will not compile.
        // See Verilog LRM 1364-2001 Section 17.2.8 for details on the
        // format of this file.

        initial
        begin
                $readmemh("init/conv1_k_g0.txt", rom);
        end

        always @ (posedge clock)
        begin
                q_a <= rom[address_a];
                q_b <= rom[address_b];
        end

endmodule


module fc_g0_mem ( input [8:0] address_a, address_b,
                                            input clock,
                                            output reg [15:0] q_a, q_b);

        // Declare the ROM variable
        reg [15:0] rom[511:0];

        // Initialize the ROM with $readmemb.  Put the memory contents
        // in the file dual_port_rom_init.txt.  Without this file,
        // this design will not compile.
        // See Verilog LRM 1364-2001 Section 17.2.8 for details on the
        // format of this file.

        initial
        begin
                $readmemh("init/fc_g0.txt", rom);
        end

        always @ (posedge clock)
        begin
                q_a <= rom[address_a];
                q_b <= rom[address_b];
        end

endmodule


module conv2_k_g4_mem ( input [7:0] address_a, address_b,
                                            input clock,
                                            output reg [15:0] q_a, q_b);

        // Declare the ROM variable
        reg [15:0] rom[255:0];

        // Initialize the ROM with $readmemb.  Put the memory contents
        // in the file dual_port_rom_init.txt.  Without this file,
        // this design will not compile.
        // See Verilog LRM 1364-2001 Section 17.2.8 for details on the
        // format of this file.

        initial
        begin
                $readmemh("init/conv2_k_g4.txt", rom);
        end

        always @ (posedge clock)
        begin
                q_a <= rom[address_a];
                q_b <= rom[address_b];
        end

endmodule


//RAM module modified from quartus template
module conv1_mem ( input [8:0] address_a, address_b,
                                   input clock,
                                   input [15:0] data_a, data_b,
                                   input wren_a, wren_b,
                                   output reg [15:0] q_a, q_b);

        reg [15:0] ram[511:0];
  logic [15:0] debug195, debug196, debug197, debug198, debug199, debug200, debug229, debug231;
  logic [8:0] prev_addra, pprev_addra;
  logic [8:0] prev_addrb, pprev_addrb;
        // Port A
        always @ (posedge clock)
        begin
    pprev_addra <= address_a;
    prev_addra <= pprev_addra;
                if (wren_a)
                begin
                        ram[prev_addra] <= data_a;
                        q_a <= data_a;
                end
                else
                begin
                        q_a <= ram[address_a];
                end
        end

        // Port B
        always @ (posedge clock)
        begin
    pprev_addrb <= address_b;
    prev_addrb <= pprev_addrb;
    if (wren_b)
                begin
                        ram[prev_addrb] <= data_b;
                        q_b <= data_b;
                end
                else
                begin
                        q_b <= ram[address_b];
                end
        end

endmodule


//RAM module modified from quartus template
module conv2_mem ( input [5:0] address_a, address_b,
                                   input clock,
                                   input [15:0] data_a, data_b,
                                   input wren_a, wren_b,
                                   output reg [15:0] q_a, q_b);

        reg [15:0] ram[63:0];

        // Port A
        always @ (posedge clock)
        begin
                if (wren_a)
                begin
                        ram[address_a] <= data_a;
                        q_a <= data_a;
                end
                else
                begin
                        q_a <= ram[address_a];
                end
        end

        // Port B
        always @ (posedge clock)
        begin
                if (wren_b)
                begin
                        ram[address_b] <= data_b;
                        q_b <= data_b;
                end
                else
                begin
                        q_b <= ram[address_b];
                end
        end

endmodule


module img_mem ( input [9:0] address_a, address_b,
                                 input clock,
                                 input [15:0] data_a, data_b,
                                 input wren_a, wren_b,
                                 output reg [15:0] q_a, q_b);

        reg [15:0] ram[1023:0];
        // Port A
        always @ (posedge clock)
        begin
                if (wren_a)
                begin
                        ram[address_a] <= data_a;
                        q_a <= data_a;
                end
                else
                begin
                        q_a <= ram[address_a];
                end
        end

        // Port B
        always @ (posedge clock)
        begin
                if (wren_b)
                begin
                        ram[address_b] <= data_b;
                        q_b <= data_b;
                end
                else
                begin
                        q_b <= ram[address_b];
                end
        end

endmodule


//RAM module modified from quartus template
module p1_mem ( input [7:0] address_a, address_b,
                                   input clock,
                                   input [15:0] data_a, data_b,
                                   input wren_a, wren_b,
                                   output reg [15:0] q_a, q_b);

        reg [15:0] ram[255:0];

        // Port A
        always @ (posedge clock)
        begin
                if (wren_a)
                begin
                        ram[address_a] <= data_a;
                        q_a <= data_a;
                end
                else
                begin
                        q_a <= ram[address_a];
                end
        end

        // Port B
        always @ (posedge clock)
        begin
                if (wren_b)
                begin
                        ram[address_b] <= data_b;
                        q_b <= data_b;
                end
                else
                begin
                        q_b <= ram[address_b];
                end
        end

endmodule


//RAM module modified from quartus template
module p2_mem ( input [3:0] address,
                                   input clock,
                                   input [15:0] data,
                                   input wren,
                                   output [15:0] q);

        reg [15:0] ram[15:0];
    reg[3:0] address_reg;

        always @ (posedge clock)
        begin
                if (wren)
                begin
                        ram[address] <= data;
            address_reg <= address;
                end
        end

    assign q = ram[address_reg];

endmodule


module P1_mem_read (input logic clk, reset, enable,
                        output logic [7:0] addr0,
                        output logic done);
    //kernel row and column count
    logic [2:0] rowcount, columncount;
    //counter for position within image row
    logic [2:0] i_count;
    logic [3:0] delay;
    logic [1:0] count;

    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            addr0 <= 8'b00000000;
            i_count <= 3'b000;
            delay <= 4'b0000;
            count <= 0;
        end
        else if (enable == 1'b1 && done == 1'b0) begin
            if (delay == 4'b0000) begin
                if (i_count == 3'b111 && rowcount == 3'b100 && columncount == 3'b100) begin
                    //done with applying the kernel over an entire image row, go to the next row
                    //Move addresses 4 rows up, 11 pixels back and 1 row down, i.e. -47
                    i_count <= 3'b000;
                    columncount <= 3'b000;
                    rowcount <= 3'b000;
                    addr0 <= addr0 - 8'b00101111;
                end
                else if (rowcount == 3'b100 && columncount == 3'b100) begin
                    //done with a position for the kernel, move forward to the next position. Move 4 rows up, 4 pixels back and 1 pixel forward, i.e. -51
                    columncount <= 3'b000;
                    rowcount <= 3'b000;
                    addr0 <= addr0 - 8'b00110011;
                    i_count <= i_count + 3'b001;
                end
                else if (columncount == 3'b100) begin
                    //kernel row has been processed, move addresses one row down and 4 pixels back, i.e. + 8
                    columncount <= 3'b000;
                    rowcount <= rowcount + 3'b001;
                    addr0 <= addr0 + 8'b00001000;
                end
                else begin
                    addr0 <= addr0 + 8'b00000001;
                    columncount <= columncount + 3'b001;
                end
                if (addr0 == 8'b10001111) begin
                    count <= count + 1;
                    addr0 <= 0;
                end
            end
            else begin
                delay <= delay + 1;
            end
        end
    end
    always_comb begin
        if(addr0 == 8'b10001111 && count == 2'b10) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
    end
endmodule


// counter/addresser for pooling 1 layer output memory write (done)
module P1_mem_write (input logic clk, reset, enable,
                        output logic [7:0] addr0, addr1,
                        output logic done);

    logic [3:0] delay;

    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            addr0 <= 8'b00000000;
            addr1 <= 8'b01001000;
            delay <= 4'b0000;
        end
        else if (enable == 1'b1 && done == 1'b0) begin
            if(delay == 4'b0001) begin
                addr0 <= addr0 + 8'b00000001;
                addr1 <= addr1 + 8'b00000001;
            end
            else begin
                delay <= delay + 4'b0001;
            end
        end
    end

    always_comb begin
        if(addr1 == 8'b10001111) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
    end

endmodule


//cycle through 4x4 input images 12 times
// counter/addresser for pooling 2 layer output memory read
module P2_mem_read (input logic clk, reset, enable,
                        output logic [3:0] addr0, count,
                        output logic done);

    logic [3:0] delay;

    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            count <= 4'b0000;
            addr0 <= 4'b0000;
            delay <= 4'b0000;
        end
        else if (enable == 1'b1 && done == 1'b0) begin
            if (delay == 4'b0000) begin
                if(addr0 == 4'b1111) begin
                    count <= count + 4'b0001;
                    addr0 <= 4'b0000;
                end
                else begin
                    addr0 <= addr0 + 4'b0001;
                end
            end
            else begin
                delay <= delay + 4'b0001;
            end
        end
    end

    always_comb begin
        if(count == 4'b1011 && addr0 == 4'b1111) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
    end

endmodule


// counter/addresser for pooling 2 layer output memory write (done)
module P2_mem_write (input logic clk, reset, enable,
                        output logic [3:0] addr0,
                        output logic done);

    logic [3:0] delay;

    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            addr0 <= 4'b0000;
            delay <= 4'b0000;
        end
        else if (enable == 1'b1 && done == 1'b0) begin
            if(delay == 4'b1) begin
                addr0 <= addr0 + 4'b0001;
            end
            else begin
                delay <= delay + 4'b0001;
            end
        end
    end

    always_comb begin
        if(addr0 == 4'b1111) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
    end

endmodule


// counter/addresser for Convolution 1 layer weight memory read (done)
module conv1_k_mem_read (input logic clk, reset, enable,
                        output logic [5:0] addr0, addr1,
                        output logic done);

    logic [3:0] delay;
    //TODO
    logic [7:0] count;
    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            addr0 <= 6'b000000;
            addr1 <= 6'b011001;
            delay <= 4'b0000;
            count <= 8'b000000;
        end
        else if (enable == 1'b1 && done == 1'b0) begin
            if(delay == 4'b0000) begin
                if (addr1 == 6'b110001) begin
                  count <= count + 8'b00000001;
                  addr0 <= 0;
                  addr1 <= 6'b011001;
                end
                else begin
                  addr0 <= addr0 + 6'b000001;
                  addr1 <= addr1 + 6'b000001;
                end
            end
            else begin
                delay <= delay + 4'b0001;
            end
        end
    end

    always_comb begin
    //TODO TODO
        if(count >= 8'b10001111 || (count == 8'b10001111 && addr1 == 6'b110001)) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
    end

endmodule


// counter/addresser for Convolution 1 layer weight memory read (done)
module conv1_k_mem_read (input logic clk, reset, enable,
                        output logic [5:0] addr0, addr1,
                        output logic done);

    logic [3:0] delay;
    //TODO
    logic [7:0] count;
    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            addr0 <= 6'b000000;
            addr1 <= 6'b011001;
            delay <= 4'b0000;
            count <= 8'b000000;
        end
        else if (enable == 1'b1 && done == 1'b0) begin
            if(delay == 4'b0000) begin
                if (addr1 == 6'b110001) begin
                  count <= count + 8'b00000001;
                  addr0 <= 0;
                  addr1 <= 6'b011001;
                end
                else begin
                  addr0 <= addr0 + 6'b000001;
                  addr1 <= addr1 + 6'b000001;
                end
            end
            else begin
                delay <= delay + 4'b0001;
            end
        end
    end

    always_comb begin
    //TODO TODO
        if(count >= 8'b10001111 || (count == 8'b10001111 && addr1 == 6'b110001)) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
    end

endmodule


// counter/addresser for Convolution 1 layer output memory write (done)
module conv1_mem_write (input logic clk, reset, enable,
                        output logic [8:0] addr0, addr1,
                        output logic done);
    //Increment addresses every 25 cycles
    logic [3:0] delay;
    logic [4:0] clk_counter;

    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            addr0 <= 9'b000000000;
            addr1 <= 9'b010010000;
            clk_counter <= 5'b00000;
            delay <= 4'b0000;
        end
        else if (enable == 1'b1 && done == 1'b0) begin
            if(delay == 4'b0001) begin
                if (clk_counter == 5'b11000) begin
                    addr0 <= addr0 + 9'b0000000001;
                    addr1 <= addr1 + 9'b0000000001;
                    clk_counter <= 5'b00000;
                end
                else begin
                    clk_counter <= clk_counter + 5'b00001;
                end
            end
            else begin
                delay <= delay + 4'b0001;
            end
        end
    end

        always_comb begin
        if(addr1 == 9'b100011111) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
    end
endmodule


//Module used to provide access to 6 kernels with two counters at once
// counter/addresser for Convolution 2 layer weight memory read
module conv2_k_mem_read (input logic clk, reset, enable,
                        output logic [7:0] addr0, addr1,
                        output logic done);
    logic [3:0] delay;
    logic [5:0] count;
    logic [7:0] addrcount;
    logic [7:0] offset;

    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            delay <= 4'b0000;
            addrcount <= 8'b00000000;
            offset <= 8'b0000000;
            count <= 6'b000000;
        end
        else if (enable == 1'b1 && done == 1'b0) begin

            //when address reaches 24, count
            if(addrcount == 8'd24) begin
                addrcount <= 8'b00000000;
                if(count == 6'd63) begin
                    count <= 6'd0;
                    if(offset == 8'b00000000) begin
                        offset <= 8'd25;
                    end
                    else if(offset == 8'd25) begin
                        offset <= 8'd50;
                    end
                end
                else begin
                    count <= count + 6'd1;
                end
            end

            //else, increment by 1
            else if(delay == 4'b0000) begin
                addrcount <= addrcount + 8'b00000001;
            end
            else begin
                delay <= delay + 4'b0001;
            end
        end
    end

    always_comb begin
        if(offset == 8'd50 && count == 6'd63) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
        addr0 = addrcount + offset;
        addr1 = addrcount + 8'b01001011 + offset;
    end
endmodule


// counter/addresser for Convolution 2 layer output memory read (done)
module conv2_mem_read (input logic clk, reset, enable,
                        output logic [5:0] addr0, addr1, addr2, addr3,
                        output logic done);

    logic [1:0] count;
    logic [3:0] delay;

    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            addr0 <= 6'b000000;
            addr1 <= 6'b000001;
            addr2 <= 6'b001000;
            addr3 <= 6'b001001;
            count <= 2'b00;
            delay <= 4'b0000;
        end
        else if (enable == 1'b1 && done == 1'b0) begin
            if(delay == 4'b0000) begin
                if(count == 2'b11) begin
                    count <= 2'b00;
                    addr0 <= addr0 + 6'b001010;
                    addr1 <= addr1 + 6'b001010;
                    addr2 <= addr2 + 6'b001010;
                    addr3 <= addr3 + 6'b001010;
                end
                else begin
                    addr0 <= addr0 + 6'b000010;
                    addr1 <= addr1 + 6'b000010;
                    addr2 <= addr2 + 6'b000010;
                    addr3 <= addr3 + 6'b000010;
                    count <= count + 2'b01;
                end
            end
            else begin
                delay <= delay + 4'b0001;
            end
        end
    end

    always_comb begin
        if( addr3 == 6'b111111) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
    end
endmodule


//Go through an 8x8 output image 3 times
// counter/addresser for Convolution 2 layer output memory write
module conv2_mem_write (input logic clk, reset, enable,
                        output logic [5:0] addr0,
                        output logic [1:0] count,
                        output logic done);
    //Increment addresses every 25 cycles
    logic [4:0] clk_counter;
    logic [3:0] delay;

    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            addr0 <= 6'b000000;
            count <= 2'b00;
            clk_counter <= 5'b00000;
            delay <= 4'b0000;
        end
        else if (enable == 1'b1 && done == 1'b0) begin
            if(delay == 4'b0001) begin
                if (clk_counter == 5'b11000) begin
                    clk_counter <= 5'b00000;
                    if(addr0 == 6'b111111) begin
                        count <= count + 2'b01;
                        addr0 <= 6'b000000;
                    end
                    else begin
                        addr0 <= addr0 + 6'b000001;
                    end
                end
                else begin
                    clk_counter <= clk_counter + 5'b00001;
                end
            end
            else delay <= delay + 4'b0001;
        end
    end


    always_comb begin
        if(count == 2'b10 && addr0 == 6'b111111) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
    end

endmodule


// counter/addresser for Fully connected layer weight memory read (done)
module fc_mem_read (input logic clk, reset, enable,
                        output logic [8:0] addr0, addr1,
                        output logic done);

    logic [3:0] delay;

    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            addr0 <= 9'b000000000;
            addr1 <= 9'b011000000;
            delay <= 4'b0000;
        end
        else if (enable == 1'b1 && done == 1'b0) begin
            if(delay == 4'b0000) begin
                addr0 <= addr0 + 9'b000000001;
                addr1 <= addr1 + 9'b000000001;
            end
            else begin
                delay <= delay + 4'b0001;
            end
        end
    end

    always_comb begin
        if(addr1 == 9'b101111111) begin
            done = 1'b1;
        end
        else begin
            done = 1'b0;
        end
    end

endmodule


// counter/addresser for input image memory write (done) //NOT NEEDED
module img_mem_write (input logic clk, reset, enable, next,
                      output logic [9:0] addr0,
                      output logic done, ack);

    logic [3:0] delay;

    always_ff @(posedge clk or posedge reset) begin
        if (reset == 1'b1) begin
            addr0 <= 10'b0000000000;
            delay <= 4'b0000;
        end
        else if (enable == 1'b1 && done == 1'b0 && delay == 4'b0001) begin
            addr0 <= addr0 + 1;
            ack <= 1'b1;
        end
        else if(next == 1'b1 && ack == 1'b1) begin
            ack <= 1'b0;
        end
        else begin
            delay <= delay + 4'b0001;
        end
    end

    always_comb begin
        if(addr0 == 10'b1100001111) begin
            done = 1'b1;
        end
    end

endmodule
```