Columbia University
Fu Foundation School of Engineering and Applied Science

Acceleration of Digit Classification Using Custom CNN on a SoC FPGA

CSEE 4840 - Embedded Systems
Final Report

Prathmesh Patel (pp2870)
Vasileios Panousopoulos (vp2518)
Rishit Thakkar (rht2122)
Tharun Kumar Jayaprakash (tj2557)

Spring 2024

1 Introduction

This project presents an approach to digit classification leveraging the DE1-SoC FPGA board. The aim is to implement an embedded system for real-time digit recognition using the MNIST dataset and a simple convolutional neural network (CNN) architecture. The FPGA environment offers unique advantages for accelerating computationally intensive tasks like image processing. Through the utilization of SystemVerilog, the CNN model has been implemented in its entirety on FPGA hardware, exploiting the parallelism and inherent efficiency of hardware resources. A camera module has been integrated with the board to enable capturing real-world handwritten digits for classification directly on the FPGA. These images are then processed appropriately on hardware and the system outputs the predicted class of the digit. This project was developed with the goal of being a novel one for the Embedded Systems class, as no Machine Learning based hardware accelerator had been previously implemented.

2 Convolutional Neural Network

2.1 Model

The primary objective of the project was to implement a system that could classify handwritten digits similar to the ones found in the famous MNIST dataset. This classification task is considered rather simple meaning that even relatively shallow neural networks can achieve high classification accuracies. Considering the limited hardware resources available on an FPGA, a minimal model should be used. Inspired by the famous LeNet architecture, for this work a custom lightweight network was designed. As shown in Figure 1, the network consists of 2 convolutional layers, using the Rectified Linear Unit (ReLU) activation function, 2 max-pooling layers for downsampling and 1 final dense layer for classification.

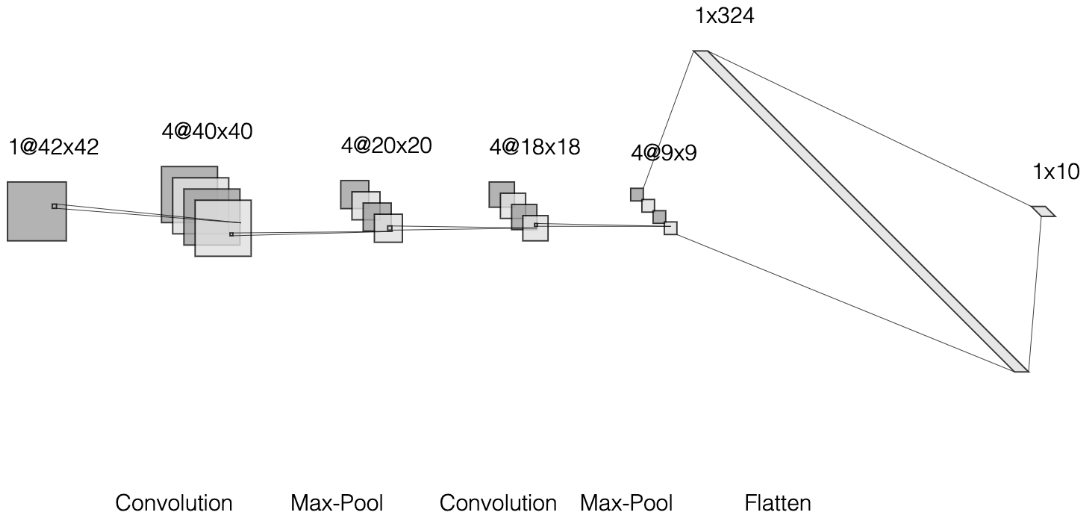

Figure 1: Model Architecture

In total, the network contains 3,438 trainable parameters (weights and biases) that need to be stored in the on-chip memory. As will be discussed in Section 3.2 the input image is sized at 42x42 pixels. This image undergoes the initial convolutional layer, yielding four output channels.

Subsequently, the image is further processed through a second convolutional layer, with four input and output channels. Both convolutional layers employ 3x3 kernels for feature extraction. Max-pooling layers, with 2x2 kernel size, are integrated into the architecture to downsample the feature maps, facilitating computational efficiency while preserving essential features. The output data of the second max-pooling layer are mapped to 324 neurons, which are processed by the fully connected (dense) layer to produce the 10 distinct classes of the dataset, that is the possibility of an image belonging to each of these classes. Throughout the architecture, ReLU is used to introduce non-linearity and enable efficient gradient propagation. This design ensures a balance between computational efficiency (resources) and performance, making it well-suited for deployment on the DE1-SoC FPGA

The model was trained in Python using Keras with the hardware constraints in mind. Specifically, the original MNIST images were resized to 42x42 pixels as the camera module deployed was expected to produce images of such resolution. Section 2.3 explains how fixed point arithmetic is used to efficiently implement the model on hardware. Given the fact that 2's complement representation is used and 8 bits are used to represent each data value, the original input image grayscale values should be modified accordingly. In particular, the 8-bit values of the input images range between 0 and 255. However, in 2's complement arithmetic 8 bits can represent values only in the interval $[\ 128, 127]$. To make training data as similar to the real data expected as possible, the MNIST images' values were scaled between 0 and 127 by applying integer division. After training, the Python model's accuracy on the test data reached 97.7%.

2.2 Building a C simulator

2.2.1 Floating Point Model

The first step towards implementing a hardware accelerator was to build a C function that would model the exact behavior of the hardware module as it would be seen from the SW perspective. For instance, the function's arguments should match the HW module's ports. In this way, the C model would be an intermediate step between Python and Verilog and it would be helpful for verification purposes.

The most critical part of the floating point simulator was to make sure that for each computation the correct data and weights were used, meaning that the correct addresses in the respective C arrays were generated. Debugging of this model required some time, but once fixed, the floating point model was able to achieve a classification accuracy of 92%. The discrepancy between the Python and C models was due to the use of double precision and single precision floating arithmetic respectively.

2.2.2 Fixed Point Model

The next steps towards hardware implementation was to define the exact arithmetic system that should be used on hardware. It is known that floating point operations are very computationally intensive and inefficient for a resource-constrained hardware platform. Therefore, the proposed implementation deployed fixed point arithmetic and specifically the Q4.4 format, meaning that the 4 MSBs of each number represent the signed integer part while the 4 LSBs correspond to the fractional part.

The process of implementing a correct fixed point model was relatively challenging as it required careful analysis of the model and intermediate results. For instance, we noticed that after training the weight values were ranging between -1.2 and 1. In fixed point notation, very small negative

numbers, i.e. 0.5, consist of multiple 1 bits, i.e. 11111000, thus when used in a multiplication the product (that needs to be truncated) usually overflows and information is lost. For this reason, it was essential to scale the original weights in the range [8, 8). Apart from easing the fixed point model implementation, this scaling improved the accuracy of the floating point simulator as well, which increased to 94.84%.

Another critical part of the fixed point model was the correct representation of the Multiplication-Accumulation (MAC) results. It is known that the product of a multiplication of two 8-bit numbers requires 16 bits to be represented accurately. In the proposed system, this 16-bit number should be truncated to 8-bit in order to be used later in other computations. For example, when a 3x3 convolution kernel operates on the image it performs 9 multiplications that are accumulated together and 1 final addition of the bias. This sequence of operation is illustrated in Figure 2.

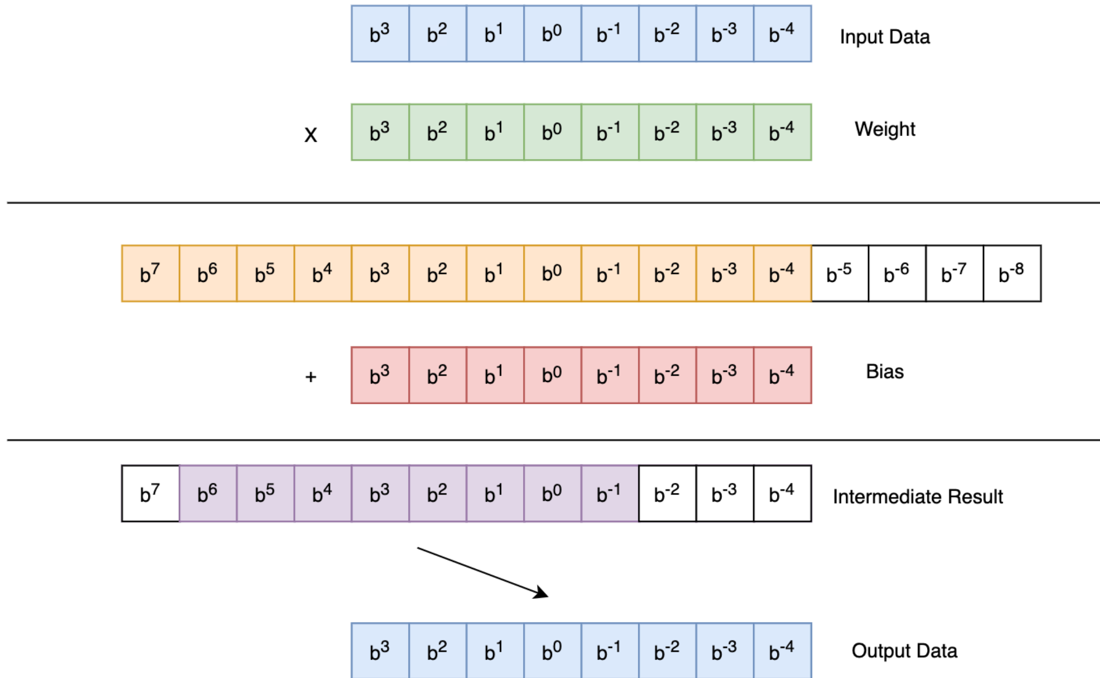

Figure 2: Bit representations during computation

As shown, the product is stored in a 16-bit number which follows the Q8.8 format. Due to rounding, by definition, the 4 LSBs must be discarded and then the bias is added. Extensive testing revealed that further shifting by 3 bits (or discarding the 3 LSBs) was required in order to avoid overflows and information loss during computations. Finally, the carefully designed fixed point model was able to achieve a classification accuracy of 90.86%.

3 System Architecture

3.1 Overview

A high level block diagram of the implemented embedded system is given in Figure 3.

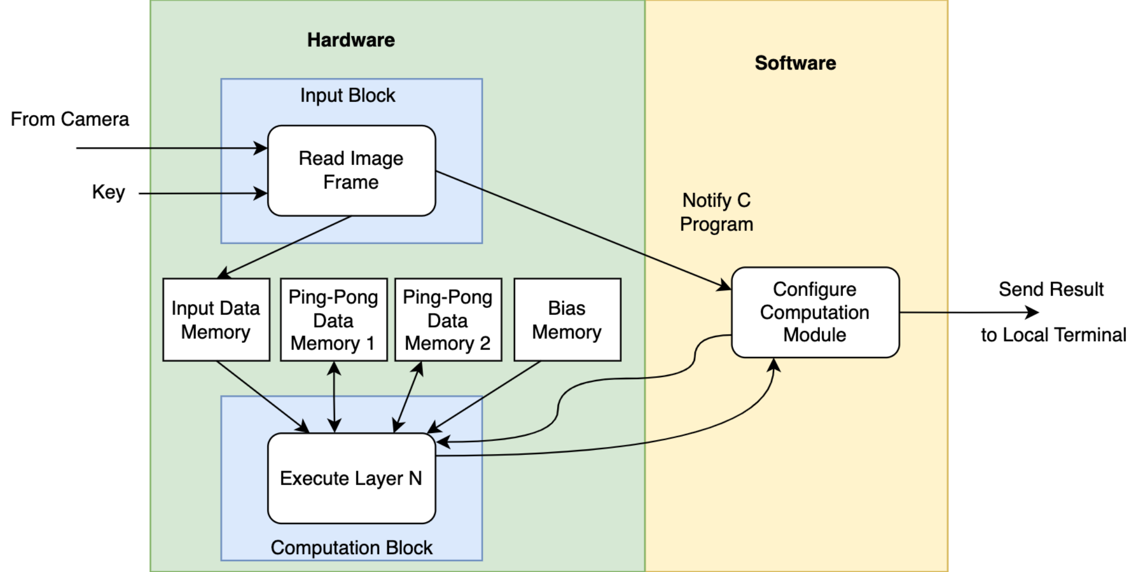

Figure 3: High level block diagram of the proposed system

First, with a key press the Input Block, implemented on hardware, is notified to store the next frame captured by the camera sensor in BRAM memory. As discussed in Section 3.2, the camera is continuously capturing new frames, but only the one that comes after the key press is stored in memory. Once the image data are available HW notifies the processor, which in turn is responsible for controlling the computation process. In particular, by passing different configuration values to the Computation Block, it specifies which of the 5 layers should be executed. Once a layer's computations are complete, the HW informs the SW, which will continue by requesting the execution of the next layers. Once the final layer is done and the 10 output values are available, SW accesses directly the BRAM that holds these values and sends to the terminal the classification result.

In order to minimize the memory resources used, the Computation Block uses two different BRAMs in a ping-pong fashion to store the results of different layers. Each layer uses the BRAM that has the output of the previous layer as input (ping) and writes its own output to the other one (pong). To simplify the design process and avoid bugs, there is one exception in this concept, as the BRAM in which the input image is stored by the Input Block is not used for ping-pong, but it's only used once, by the Computation Block to read the input data. Lastly, one BRAM is used as ROM to store the trained parameters, that are hardwired before compilation.

3.2 Camera Sensor

The OV7670 camera, shown in Figure 4, is used to acquire an image of the handwritten digit. The sensor is a lightweight module based on the Serial Camera Control Bus (SCCB) protocol, which is a subset of the I2C protocol. It has 18 pins with the configuration seen in Figure 5. Similar to I2C, SCCB has a `XCLK` signal, which is supplied by the master device to synchronize

the data transmission of the slave. In exchange, the module uses the PCLK signal, in conjunction with the D0-D7 data lines to supply an output of 8 bits. Since our application requires single-channel grayscale input, the camera is configured to the YUV422 format to output an image. The luminescence value is retrieved from the Y component of this format, and is used as the grayscale intensity of an individual pixel.

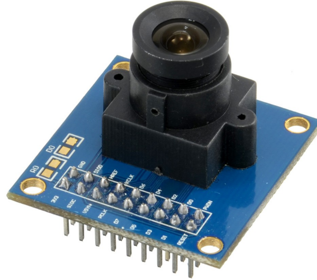

Figure 4: OV7670 Camera Module

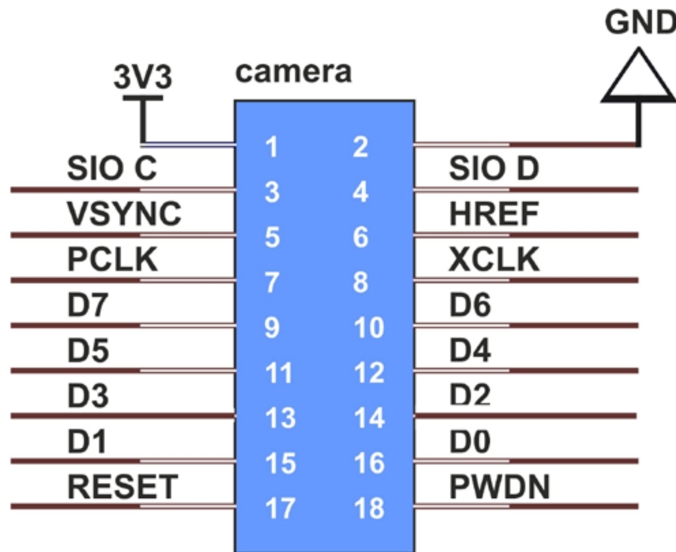

Figure 5: OV7670 Pinout

3.2.1 Camera Control and Initial Configuration

In order to minimize the number of computations, the minimum image size that can be produced by the camera module, that is 42x42 pixels, was considered. To achieve this, the DSP unit on board the OV7670 is leveraged. As seen in Figure 6, the camera is configured to output full VGA 640x480 default resolution, which is passed into the downsampling unit, reducing the resolution by 1/8. The subsequent digital zoom unit was intended to further refine the resulting 80x60 image to 42x42. However, due to an issue involving clock mismatch, this could not be achieved using the onboard image processing capability. Therefore, as discussed in further detail in the next section, a manual cropping feature was added to achieve this.

A total of 78 registers were configured on the OV7670 sensor to capture and read an image, involving the configuration of VGA resolution, YUV image output format, resolution reduction,

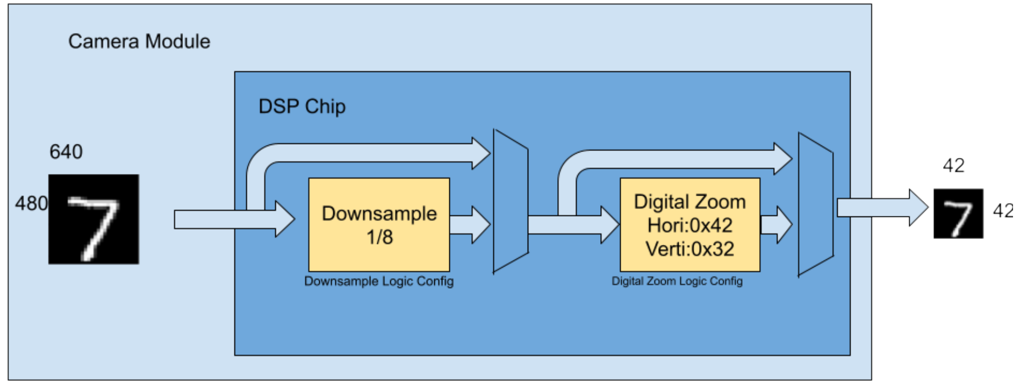

Figure 6: Resize Logic in OV7670

and the appropriate internal clock settings to facilitate these customizations. An important aspect to note is that, when implementing resolution changes, the PCLK signal needed to be adjusted specially to get the correct synchronization and result in the desired output format. Starting from full VGA resolution, if the size was reduced by 1/2, 1/4, or 1/8, the PCLK signal also needed to be adjusted by 1/2, 1/4, or 1/8. This signal further needed to be tweaked if the additional digital zoom unit was used to achieve a custom final size, different from the provided scaling factors. Some of the key register descriptions and values are shown in the Table 1. A full list of registers can be located in the camera_interface.sv file in the code listing.

Register	Description	Value
COM 3 (0x0C)	Primarily for enabling scaling and downward sampling, but has additional parameter settings	0x0C
CLKRC (0x11)	Slowing down incoming clock (can keep default)	0x60
COM 7 (0x12)	Output format configuration	0x00
COM 14 (0x3E)	PCLK configuration (change with res change)	0x1b
SCALING XSC (0x70)	Digital zoom x config (will need to change PCLK if modified)	not used
SCALING YSC (0x71)	Digital zoom y config (will need to change PCLK if modified)	not used
SCALING DCWCTR (0x72)	Reduces size from default VGA 640x480	0x77
SCALING PCLK DIV (0x73)	PCLK divider. Change when size reduced using 0x72 or other register	0x03

Table 1: Key register listing

3.2.2 Image Retrieval, Cropping, and Storage

Upon startup, the camera is first configured with custom register values and prepared for image retrieval. The data will arrive in the sequence shown in Figure 7. The U and V fields are shared between every two pixels; therefore, to reconstruct a pixel into RGB values, the operation will need to be done in pairs. However, as the model only requires grayscale images, Y values can be sampled without any further calculation. The high-level VGA timing is shown in Figure 8. It can be observed that the tp is twice the time of tpclk, due to the shared use of U and V fields between adjacent pixels. From this incoming data stream, every alternate byte can be stored, corresponding to the Y value of every pixel.

Upon the detection of VSYNC falling edge and HREF being high, every rising edge of PCLK is an indication of a byte of data coming in. Every such alternate byte is captured and stored in a register. Once the second byte passes, which is meant to be ignored for sampling just the Y values, the data from the register is written into a BRAM block. Once VSYNC is detected to go high again, the transmission of a full frame has been completed, and the next fame can be ingested.

INPUT	DATA
1	U0
2	Y0
3	V0
4	Y1
5	U1
6	Y2
7	V2
8	Y3
9	U2
...	...

Figure 7: YUV Data Arrival Sequence

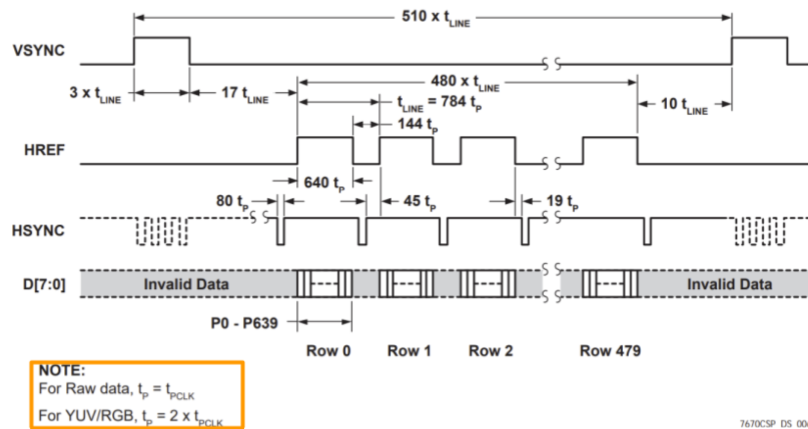

Figure 8: VGA Timing Based on OV7670 Datasheet

To implement the cropping feature, only the first 42 bytes of every row are stored, and rows after row 42 are ignored. This is tracked using counters created in SystemVerilog. Although this results in an image which is 42x42, it limits the image scope to the top-left portion of the incoming frame from the camera.

An additional consideration is that the camera continuously sends frames at 30fps (depending on the resolution and color configuration), but the custom-created logic only captures one incoming frame upon receiving external input from a user, in the form of a button press. When this button press is detected and a new incoming frame transmission is detected, the incoming luminescence values are processed and stored in a BRAM block to be used by the accelerator and VGA driver. A complete depiction of this workflow is shown in Figure 12.

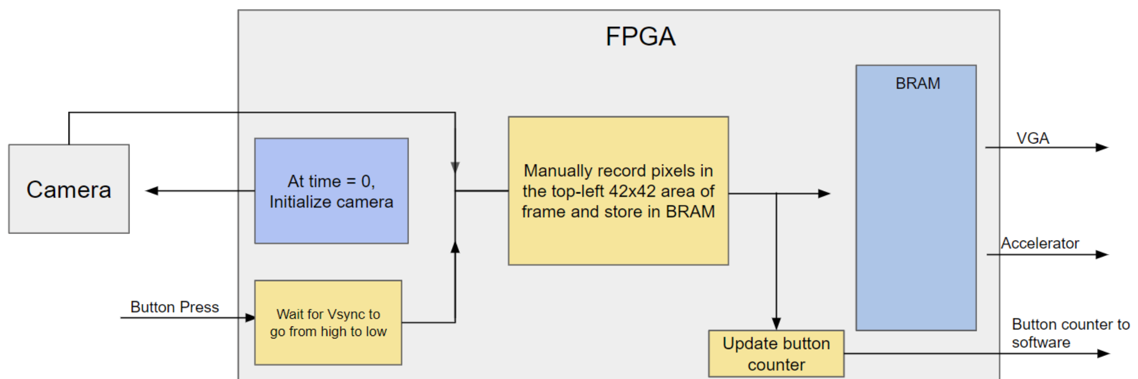

Figure 9: Camera Work ow

3.3 Hardware Accelerator

As shown in Figure 3, all the computations required by the CNN are performed by the Computation Block, that is implemented in its entirety on hardware with SystemVerilog. A high level block diagram of the accelerator is given in Figure 10.

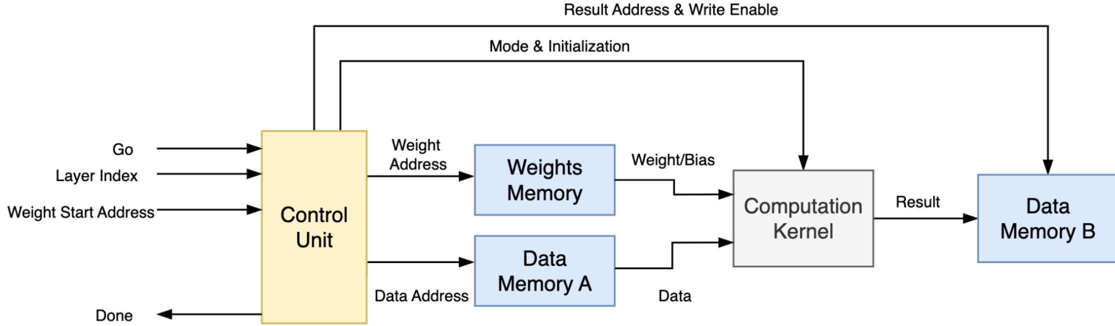

Figure 10: High level block diagram of the CNN accelerator

3.3.1 Control Unit

The most critical component of the accelerator is the Control Unit (CU), which is responsible for both the communication with software and the control of the dataflow within the HW module. The accelerator has been designed in a parameterised fashion, as the SW can select which layer needs to be executed by specifying the `Layer Index` register. Once the desired value is given as well as the respective initial memory address at which the block of weights used by the layer is stored, the SW can invoke the accelerator by asserting the `Go` signal. The CU is implemented as a Finite State Machine (FSM) of 21 states that contains 5 different state "paths" that corresponds to the 5 different layers that are supported. Once the `Go` is high, the CU initializes the Computation Kernel and initiates the correct address generation sequence. The most critical point of the CU is ensuring the correct timing and synchronization between the Computation Kernel and the memories. Synchronization means that the Computation Kernel is initialized in the correct cycle, the correct addresses are generated at each cycle for data and weights and that the **write enable** signal of the output memory is asserted only at the cycle when a valid result is available. Once the computations are complete, the Control Unit asserts the `Done` signal to notify the SW. In terms of FPGA resources, the Control Unit was implemented with logic cells and both ping-pong memories were mapped to 8KB BRAMs.

3.3.2 Computation Kernel

The kernel's block diagram is given in Figure 11.

By definition, a CNN performs two different computations, MAC and max-pooling. The proposed kernel was designed to implement both computations in parallel. On the one hand, a MAC unit performs consecutive operations on new data and weights that are coming in consecutive cycles. Apparently, within the MAC unit a register exists to hold the intermediate results. As explained in Section 2.2.2, the result is truncated to an 8-bit value, which undergoes the ReLU activation. This function can be modeled by a MUX that selects either the original number or 0 depending on the Sign Bit (MSB) of the number. On the other hand, a 2-number comparator is designed that receives data in consecutive cycles and only stores the maximum value seen.

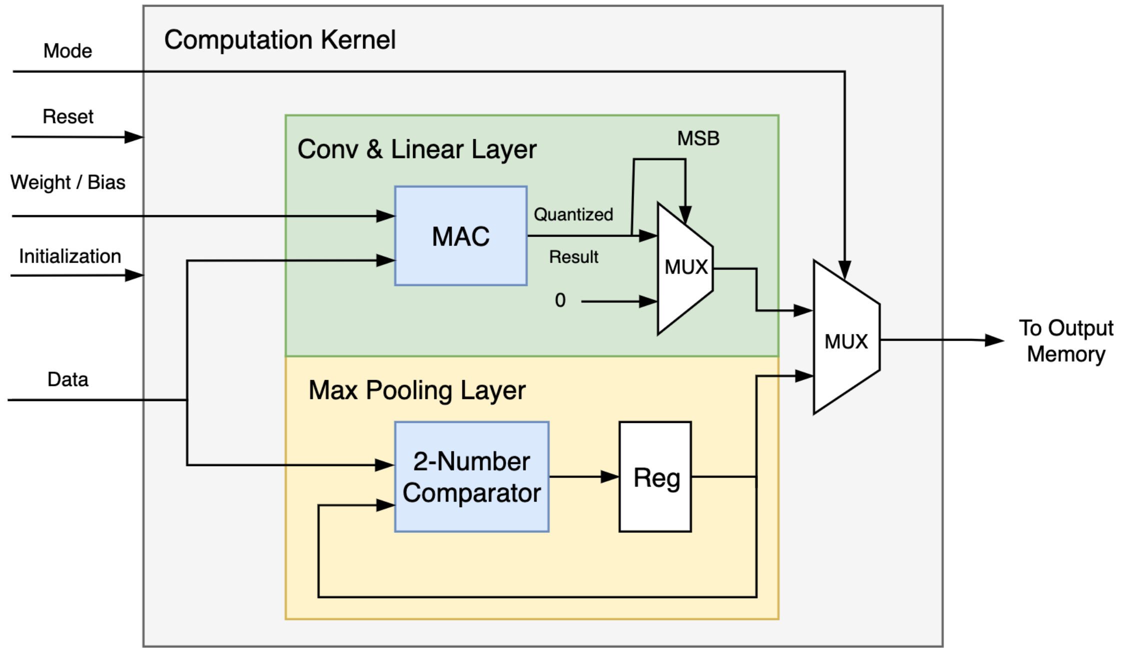

Figure 11: Block diagram of the Computation Kernel

Although both computations are always performed, the kernel's output depends on the Mode signal that controls the output MUX and is driven by the CU. When a convolutional or dense layer is executed the signal is low, but when a max-pooling layer is running the signal is set high. Proper behavior is ensured if the CU drives both the initialization and the write enable signals correctly.

The biggest challenge while designing this module was to implement the 2's complement multiplication and addition correctly. Moreover, the need for detecting negative numbers for the ReLU function required appropriate detection and handling of overflow. Lastly, it is important to note that the MAC unit was mapped to a DSP block as expected, to improve efficiency.

3.3.3 Verification

Due to the high complexity of the CNN accelerator, it was essential to carefully verify that each computation step produced the correct results. The C Simulator discussed in Section 2.2 eased the process significantly. In particular, the "golden" results of each layer in the C fixed-point program were extracted and were compared against the results of the HW module. Initially, the entire process was performed using an RTL testbench (RTL verification) in ModelSim. Once the accelerator was verified, a fully automatic method was implemented by using Verilator and writing a C++ testbench. This testbench, which models how SW "sees" the HW module, automated the verification process and the required output text file comparisons and was essential to perform fast verification when changes on the RTL were required during the integration phase.

4 The Hardware/Software Interface

The hardware-software interface is primarily based on the Intel Avalon Bus protocol, comprised of `ADDRESS`, `CHIPSELECT`, `WRITE`, `WRITEDATA`, `READ`, and `READDATA` signals. This interface connects the C-based software running on Linux with the programmable logic block of the FPGA. A complete workflow of the system is illustrated in figure 12. Two 8-bit registers, A and B, are used to read data into the software, and two additional registers, C and D, are used to write data from the software to the hardware. The camera workflow, illustrated above, comprises the image capture block. As mentioned earlier, upon button press, an incoming image is processed and stored in the Input Image BRAM. From this block, a simple VGA driver continuously reads the data and displays the image on a monitor.

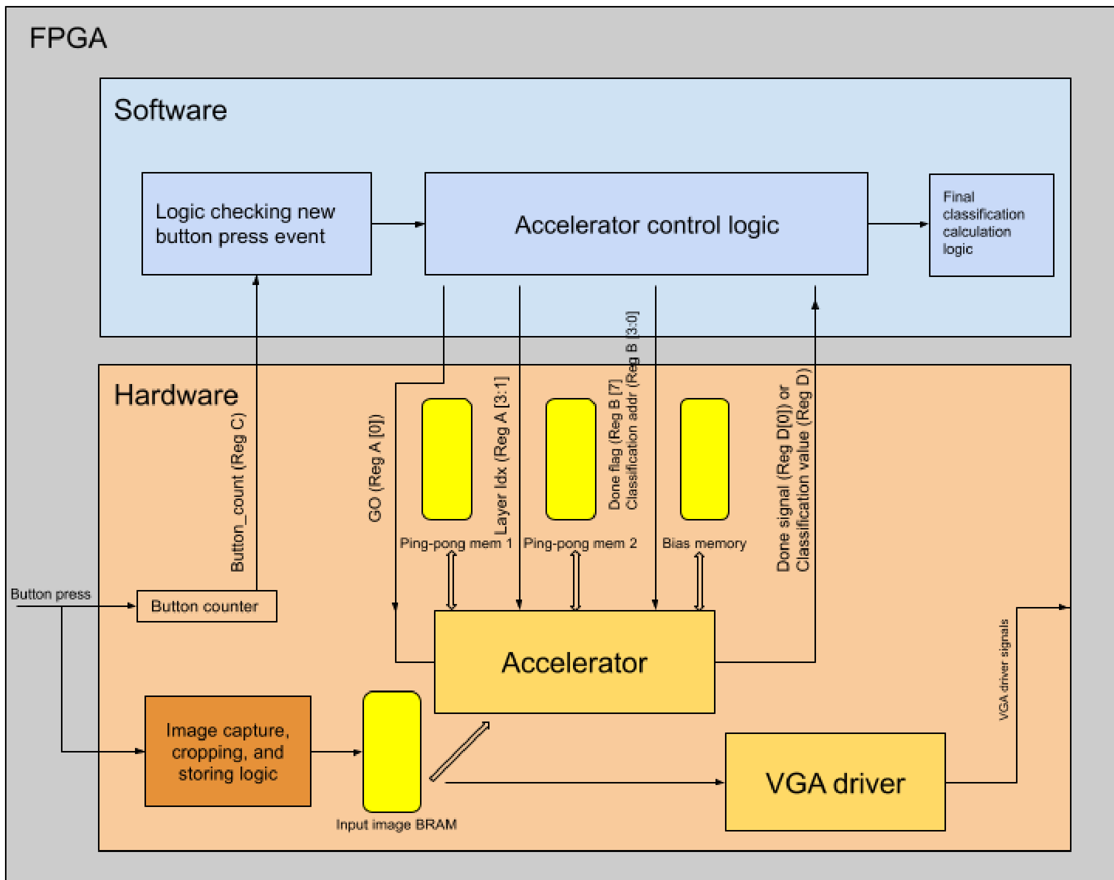

Figure 12: HW-SW Interface Work ow

When the image is stored, a button counter is updated. This value is written on the read bus through register C, when requested from the software. The software continuously polls this value and compares it to the previous valid value. If a change is detected, this is considered as a trigger for the accelerator. The `Go` and `Layer Index` signals are then sent to the hardware, combined in register A. The LSB corresponds to the `Go` signal, while `Layer Index` is stored in A[3:1].

In order to retrieve the `Done` signal from register D when the layer is done, the MSB of register B, called `Done Flag`, is set to 0. This indicates the hardware to output a `Done` signal upon the completion of the computation for each layer. This process, of passing the `Go` and `Layer Index` values to HW and reading back the `Done` bit, repeats five times for the five layers of the network.

After the last done signal is received, SW needs to read the 10 output values that correspond to the output classes. To do that, register B is used to pass a different message to hardware. The MSB (`Done Flag`) is set to 1, meaning that a classification result needs to be read and not the `Done` signal. Also, the respective address of the output memory to be read is passed throuth the 4 LSBs (B[3:0]) of the register. Once the HW sees the request, it accesses the output memory and sends back to SW the corresponding prediction value on register D. This process is repeated 10 times and the software outputs the maximum value read as the prediction class for the captured image.

5 Compromises and Limitations

During the design and integration phases, some roadblocks led us to compromise on our initial target system, and adopt alternate approaches. Firstly, as explained in Section 3.2.1 the digital zoom block of the camera could not be configured exactly to generate the necessary clock and reduce the rectangular 80x60 image to a square 42x42 image. A manual cropping solution was used as an alternative for this problem.

Secondly, whilst configuring the Avalon Bus read and write interface to add more registers, an error suggesting insufficient FPGA resources was generated. Despite trying several different configurations, the error persisted. Therefore, only two read and two write registers were used, instead of having different registers for each signal. A partial solution to this problem was to combine the essential signals in different bits of the available registers, as described in Section 4.

The impact of this limitation on the system capabilities was twofold. On the one hand, the `Weight Start Address` signal was not implemented as shown in Figure 10. This value corresponds to the memory address where the block of weights for a specific layer starts. Since it was not possible to pass this value through software, a minor modification to the Control Unit was needed, as these value were hardwired internally in a register depending on the `Layer Index`. On the other hand, it was not possible to support real-time comparison with the SW model. In particular, if more registers were available, the input image data captured would be read from SW and passed to the C simulator to compare the HW output with the golden output. Note, that the correct functionality of the accelerator is already fully verified, but it would be interesting to be able to see the simulator results when the system fails to classify a digit correctly.

Another point to mention is that the user is expected to draw a digit in white color on a black background. This because the MNIST images are originally in this format and the system was designed to match this behavior. However, with some preprocessing it would be possible to invert the colors so as to make it easier for the user, as they would be required to draw black digits on white background, which is easily done by using a blank piece of paper. The current implementation can be demoed by using a smartphone to draw white digits on black background.

Lastly, it was observed that the Verilog array used to store the input image was implemented using 2 BRAMs instead of 1. This was due to the fact, that in the given code this memory was read both sequentially (by the accelerator) and combinationally (by the VGA module). Although the total resource consumption was well below the max limit, this was an inefficiency in the final implementation and could have been avoided with minor modifications.

6 Code Listing

6.1 QSYS Component Top Module

```
module vga_ball(
    input logic clk,
    input logic  reset,
    input logic [7:0] writedata,
    input logic  write,
    input   chipselect,
    input logic [2:0] address,
    output logic [7:0] readdata,
    input logic  read,

    ////////////////BUTTON/////////////////
    input wire [3:0] VGA_KEY,

    /////////////////LEDS//////////////////
    output wire [9:0] VGA_LEDR,

    /////////////////CAMERA///////////////////
    input wire VGA_CMOS_PCLK,VGA_CMOS_HREF,VGA_CMOS_VSYNC,
    input wire [7:0] VGA_CMOS_DB,
    inout VGA_CMOS_SDA,VGA_CMOS_SCL,

    output wire VGA_CMOS_RST_N, VGA_CMOS_PWDN, VGA_CMOS_XCLK,
    output wire VGA_CMOS_PWR, VGA_CMOS_GND,

    /////////////////VGA//////////////////
    output logic [7:0] VGA_R, VGA_G, VGA_B,
    output logic  VGA_CLK, VGA_HS, VGA_VS
    );

    wire[15:0] dout;
    wire[15:0] din;
    wire empty_fifo;
    wire state;
    wire rd_en;
    wire rd_en_cam;
    wire rd_en_to_avalon;

    reg [27:0] c;
    wire[7:0] button_count;
    wire[7:0] pixel_data;
    reg[15:0] read_ptr_avalon;
```

```
//cases for memory read inquiry coming from software
localparam BUTTON_COUNT = 0,
           PIXEL_DATA = 1,
           LAYER_DONE = 2;

localparam WRITE_PIXEL_LOW = 0,
           WRITE_PIXEL_UP = 1,
           WRITE_GO = 2,
           WRITE_LAYER_IDX = 3,
           WRITE_DATA_ADDR_LOW = 4,
           WRITE_DATA_ADDR_UP = 5,
           WRITE_WEIGHT_ADDR_LOW = 6,
           WRITE_WEIGHT_ADDR_UP = 7;

logic read_flag;
assign read_flag = 1;

always @(posedge clk) begin
    if (reset) begin
       read_ptr_avalon <= 8'h0;
       readdata <= 8'h0;
    end else if (chipselect && write) begin
        case (address)
           WRITE_PIXEL_LOW : read_ptr_avalon[7:0] <= writedata;
           WRITE_PIXEL_UP : read_ptr_avalon[15:8] <= writedata;
        endcase
    end else if (chipselect && read) begin
        case (address)
           BUTTON_COUNT : readdata <= button_count;
           PIXEL_DATA : readdata <= pixel_data;
        endcase
    end
end

always @(posedge clk) begin
    if(reset)
        c <= 0;
    else begin
        if (c[27]) begin
            VGA_LEDR[3] <= 0;
            VGA_LEDR[4] <= 0;
        end else begin
            VGA_LEDR[3] <= 1;
            VGA_LEDR[4] <= 1;
        end
        c <= c + 1;
```

```
    end
end

wire pir;
assign pir = 0;
assign VGA_CMOS_PWR = 1;
assign VGA_CMOS_GND = 0;
assign VGA_LEDR[8] = !(reset);

logic [15:0] memory_idx;
assign memory_idx = (read_flag) ? read_ptr_avalon : image_idx[15:0];


//control logic for retrieving data from camera, storing data to asyn_fifo,
    and sending data to sdram
camera_interface m0
(
    .clk(clk),
    .clk_100(clk),
    .clk_vga(VGA_CLK),
    .rst_n(!reset),
    .key(VGA_KEY),
    .pir(pir),
    .rd_en_vga(rd_en),
    .rd_en(rd_en),
    .dout_vga(din),
    .button_count_q(button_count),

    //camera pinouts
    .cmos_pclk(VGA_CMOS_PCLK),
    .cmos_href(VGA_CMOS_HREF),
    .cmos_vsync(VGA_CMOS_VSYNC),
    .cmos_db(VGA_CMOS_DB),
    .cmos_sda(VGA_CMOS_SDA),
    .cmos_scl(VGA_CMOS_SCL),
    .cmos_rst_n(VGA_CMOS_RST_N),
    .cmos_pwdn(VGA_CMOS_PWDN),
    .cmos_xclk(VGA_CMOS_XCLK),

    //memory to avalon
    .rd_en_to_avalon(rd_en_to_avalon),
    .pixel_data(pixel_data),
    .rd_ptr_from_avalon(memory_idx),

    //Debugging
    .led_start(VGA_LEDR[2]),
```

```
    .flag(VGA_LEDR[0]),
    .flag2(VGA_LEDR[1])
);

//control logic for retrieving data from sdram, storing data to asyn_fifo,
    and sending data to vga
vga_interface m2
(
    .clk(clk),
    .rst_n(!reset),

    //asyn_fifo IO
    .din(din),
    .clk_vga(VGA_CLK),
    .rd_en(rd_en),

    //VGA output
    .vga_out_r(VGA_R),
    .vga_out_g(VGA_G),
    .vga_out_b(VGA_B),
    .vga_out_vs(VGA_VS),
    .vga_out_hs(VGA_HS)
);


logic [7 : 0] image;
logic [31 : 0] image_idx;

assign image = pixel_data;

// Software
logic go; //write
logic [ 2 : 0] layer_index; //write
logic [15 : 0] data_address; //write
logic [31 : 0] data_size; //not needed
logic [15 : 0] weight_address; //write
logic [31 : 0] weight_size; //not needed
logic [31 : 0] result_address; //not needed
logic [ 2 : 0] done; //read

assign data_size = 0;
assign weight_size = 0;
assign result_address = 0;

logic rstn;
assign rstn = ~reset;
```

```
    top top_acc
    (
        .clk(clk),
        .rstn(rstn),
        .image(image),
        .image_idx(image_idx),
        .go(go),
        .layer_index(layer_index),
        .data_address({{16{1'b0}}, data_address}),
        .data_size(data_size),
        .weight_address({{16{1'b0}},weight_address}),
        .weight_size(weight_size),
        .result_address(result_address),
        .done(done)
    );

endmodule
```

6.2 Camera Interfacre

```
module camera_interface (
    input wire clk,
    input wire clk_100,
    input wire clk_vga,
    input wire rst_n,
    input wire[3:0] key,
    input wire pir,
    input wire rd_en,
    input wire rd_en_vga,

    //camera pinouts
    input wire cmos_pclk,
    input wire cmos_href,
    input wire cmos_vsync,
    input wire[7:0] cmos_db,
    inout cmos_sda,
    inout cmos_scl, //i2c comm wires
    output wire cmos_rst_n,
    output wire cmos_pwdn,
    output wire cmos_xclk,
    output wire empty,
    output reg[7:0] button_count_q,

    //memory to avalon
    input wire rd_en_to_avalon,
    input wire [12:0] rd_ptr_from_avalon,
    output wire [7:0] pixel_data,


    //Debugging
    output led,
    output wire led_start, flag, flag2,

    /////////////////////////////////////////////////////
    output wire[7:0] dout_vga
    );

    //FSM state declarations
    localparam idle = 0,
               start_sccb = 1,
               write_address = 2,
               write_data = 3,
               digest_loop = 4,
               delay = 5,
```

```
            vsync_fedge = 6,
            byte1 = 7,
            byte2 = 8,
            fifo_write = 9,
            stopping = 10;

localparam wait_init = 0,
            sccb_idle = 1,
            sccb_address = 2,
            sccb_data = 3,
            sccb_stop = 4;

localparam rest = 0,
            vsync_fedge_SD = 1,
            byte1_SD = 2,
            byte2_SD = 3;


localparam MSG_INDEX = 78;
//number of the last index to be digested by SCCB

reg[3:0] state_q = 0, state_d;
reg[2:0] sccb_state_q = 0, sccb_state_d;
reg[7:0] addr_q, addr_d;
reg[7:0] data_q, data_d;
reg start, stop;
reg[7:0] wr_data;
wire rd_tick;
wire[1:0] ack;
wire[7:0] rd_data;
wire[3:0] state;
reg[3:0] led_q = 0, led_d;
reg[27:0] delay_q = 0, delay_d;
reg start_delay_q = 0, start_delay_d;
reg delay_finish;
reg[15:0] message[80:0];
reg[7:0] message_index_q = 0, message_index_d;
reg[15:0] pixel_q, pixel_d;
reg wr_en;

wire key0_tick, key1_tick, key2_tick, key3_tick;

//buffer for all inputs coming from the camera
reg pclk_1, pclk_2, href_1, href_2, vsync_1, vsync_2;

//////////////////////////////////////////////////
```

```
reg[2:0] lines_q, lines_d;
reg[18:0] count_q = 0, count_d;
wire full;
reg[12:0] wr_ptr_q; //binary counter for write pointer
wire[12:0] wr_ptr_d;
reg[12:0] wr_ptr_real_q; //binary counter for write pointer
wire[12:0] wr_ptr_real_d;
reg[12:0] rd_ptr_q = 0; //binary counter for read pointer
wire[12:0] rd_ptr_d;
wire start_storing_d, delay_storing_d;
reg start_storing_q, delay_storing_q;
wire state_42_d, line_counter_d;
wire[7:0] button_count_d;
reg state_42_q, line_counter_q;
reg[7:0] addr_avalon;

/////////////////////////////////////////////////

//collection of all adddresses and values to be written in the camera
always_comb begin

    message[0]=16'h12_80; //reset all register to default values
    message[1]=16'h12_04; //set output format to RGB
    message[2]=16'h15_00; //pclk will not toggle during horizontal blank
    message[3]=16'h40_d0; //RGB565

    // These are values scalped from
        https://github.com/jonlwowski012/OV7670_NEXYS4_Verilog/blob/master/ov7670_registers_v
    message[4]= 16'h12_00; // COM7, set RGB color output
    message[5]= 16'h11_60; // CLKRC internal PLL matches input clock
    message[6]= 16'h0C_04; // COM3, default settings
    message[7]= 16'h3E_1b; // COM14, no scaling, normal pclock
    message[8]= 16'h04_00; // COM1, disable CCIR656
    message[9]= 16'h40_c0; //COM15, RGB565, full output range
    message[10]= 16'h3a_04; //TSLB set correct output data sequence (magic)
    message[11]= 16'h14_6A; //COM9 MAX AGC value x4 0001_1000
    message[12]= 16'h42_B0; //MTX1 all of these are magical matrix
        coefficients
    message[13]= 16'h50_80; //MTX2write
    message[14]= 16'h51_00; //MTX3
    message[15]= 16'h52_22; //MTX4
    message[16]= 16'h53_5e; //MTX5
    message[17]= 16'h54_80; //MTX6
    message[18]= 16'h58_9E; //MTXS
    message[19]= 16'h3D_40; //Cstart_delay_dOM13 sets gamma enable
    message[20]= 16'h17_14; //HSTART start high 8 bits
```

```
message[21]= 16'h18_02; //HSTOP stop high 8 bits //these kill the odd
    colored line
message[22]= 16'h32_80; //HREF edge offset
message[23]= 16'h19_03; //VSTART start high 8 bits
message[24]= 16'h1A_7B; //VSTOP stop high 8 bits
message[25]= 16'h03_0A; //VREF vsync edge offset
message[26]= 16'h0F_41; //COM6 reset timings
message[27]= 16'h1E_20; //MVFP disable mirror / flip //might have magic
    value of 03
message[28]= 16'h33_0B; //CHLF //magic value from the internet
message[29]= 16'h3C_78; //COM12 no HREF when VSYNC low
message[30]= 16'h69_00; //GFIX fix gain control
message[31]= 16'h74_00; //REG74 Digital gain control
message[32]= 16'hB0_84; //RSVD magic value from the internet *required*
    for good color
message[33]= 16'hB1_0c; //ABLC1
message[34]= 16'hB2_0e; //RSVD more magic internet values
message[35]= 16'hB3_80; //THL_ST
message[36]= 16'h70_20;
message[37]= 16'h71_20;
message[38]= 16'h72_77;
message[39]= 16'h73_03;
message[40]= 16'ha2_01;
//gamma curve values
message[41]= 16'h7a_20;
message[42]= 16'h7b_10;
message[43]= 16'h7c_1e;
message[44]= 16'h7d_35;
message[45]= 16'h7e_5a;
message[46]= 16'h7f_69;
message[47]= 16'h80_76;
message[48]= 16'h81_80;
message[49]= 16'h82_88;
message[50]= 16'h83_8f;
message[51]= 16'h84_96;
message[52]= 16'h85_a3;
message[53]= 16'h86_af;
message[54]= 16'h87_c4;
message[55]= 16'h88_d7;
message[56]= 16'h89_e8;
//AGC and AEC
message[57]= 16'h13_e0; //COM8, disable AGC / AEC
message[58]= 16'h00_00; //set gain reg to 0 for AGC
message[59]= 16'h10_00; //set ARCJ reg to 0
message[60]= 16'h0d_40; //magic reserved bit for COM4
message[61]= 16'h14_18; //COM9, 4x gain + magic bit
```

```
    message[62]= 16'ha5_05; // BD50MAX
    message[63]= 16'hab_07; //DB60MAX
    message[64]= 16'h24_95; //AGC upper limit
    message[65]= 16'h25_33; //AGC lower limit
    message[66]= 16'h26_e3; //AGC/AEC fast mode op region
    message[67]= 16'h9f_78; //HAECC1
    message[68]= 16'ha0_68; //HAECC2
    message[69]= 16'ha1_03; //magic
    message[70]= 16'ha6_d8; //HAECC3
    message[71]= 16'ha7_d8; //HAECC4
    message[72]= 16'ha8_f0; //HAECC5
    message[73]= 16'ha9_90; //HAECC6
    message[74]= 16'haa_94; //HAECC7
    message[75]= 16'h13_e5; //COM8, enable AGC / AEC
    message[76]= 16'h1E_23; //Mirror Image
    message[77]= 16'h69_06; //gain of RGB(manually adjusted)
    message[78]= 16'h6b_50;
end

//register operations
always @(posedge clk_100,negedge rst_n) begin
    if (!rst_n) begin
        state_q <= 0;
        led_q <= 0;
        delay_q <= 0;
        start_delay_q <= 0;
        message_index_q <= 0;
        pixel_q <= 0;

        sccb_state_q <= 0;
        addr_q <= 0;
        data_q <= 0;
        delay_storing_q <= 0;
        start_storing_q <= 0;
        button_count_q <= 0;
        line_counter_q <= 1;
        state_42_q <= 0;
    end else begin
        state_q <= state_d;
        delay_q <= delay_d;
        start_delay_q <= start_delay_d;
        message_index_q <= message_index_d;
        pclk_1 <= cmos_pclk;
        pclk_2 <= pclk_1;
        href_1 <= cmos_href;
        href_2 <= href_1;
```

```
        vsync_1 <= cmos_vsync;
        vsync_2 <= vsync_1;
        pixel_q <= pixel_d;
        sccb_state_q <= sccb_state_d;
        addr_q <= addr_d;
        data_q <= data_d;
        delay_storing_q <= delay_storing_d;
        start_storing_q <= start_storing_d;
        button_count_q <= button_count_d;
        line_counter_q <= line_counter_d;
        state_42_q <= state_42_d;
    end
end

//FSM next-state logics
always @* begin
    state_d = state_q;
    led_d = led_q;
    start = 0;
    stop = 0;
    wr_data = 0;
    start_delay_d = start_delay_q;
    delay_d = delay_q;
    delay_finish = 0;
    message_index_d = message_index_q;
    pixel_d = pixel_q;
    wr_en = 0;
    wr_ptr_d = wr_ptr_q;
    wr_ptr_real_d = wr_ptr_real_q;

    sccb_state_d = sccb_state_q;
    addr_d = addr_q;
    data_d = data_q;


    //delay logic
    if (start_delay_q)
            delay_d=delay_q + 1'b1;

    //delay between SCCB transmissions (0.66ms)
    if (delay_q[16] && message_index_q != (MSG_INDEX + 1) && (state_q !=
        start_sccb)) begin
        delay_finish = 1;
        start_delay_d = 0;
        delay_d = 0;
```

```
    //delay BEFORE SCCB transmission, AFTER SCCB transmission, and BEFORE
        retrieving pixel data from camera (0.67s)
    end else if ((delay_q[26] && message_index_q == (MSG_INDEX + 1)) ||
        (delay_q[26] && state_q == start_sccb)) begin
        delay_finish = 1;
        start_delay_d = 0;
        delay_d = 0;
    end

    if (!pir) begin
        case(state_q)

////////Begin: Setting register values of the camera via SCCB///////////
            //idle for 0.6s to start-up the camera
            idle: begin
                if (delay_finish) begin
                    state_d = start_sccb;
                    start_delay_d = 0;
                        end else begin
                    start_delay_d = 1;
                    led_start = 1;
                end
            end

            //start of SCCB transmission
            start_sccb: begin
                start = 1;
                wr_data = 8'h42; //slave address of OV7670 for write
                state_d = write_address;
            end

            write_address: begin
                if (ack == 2'b11) begin
                    wr_data = message[message_index_q][15:8]; //write address
                    state_d = write_data;
                    led_start = 0;
                end
            end

            write_data: begin
                if (ack == 2'b11) begin
                    wr_data = message[message_index_q][7:0]; //write data
                    state_d = digest_loop;
                            end
            end
```

```
        digest_loop: begin
            if (ack == 2'b11) begin //stop sccb transmission
                stop = 1;
                start_delay_d = 1;
                message_index_d = message_index_q + 1'b1;
                state_d = delay;
            end
        end

        delay: begin
            //if all messages are already digested, proceed to retrieving
                camera pixel data
            if (message_index_q == (MSG_INDEX + 1) && delay_finish) begin
                state_d = vsync_fedge;
                led_d = 4'b0110;
                led_start=0;
            end else if (state == 0 && delay_finish)
                //small delay before next SCCB transmission(if all
                    messages are not yet digested)
                state_d=start_sccb;
        end

///////////////Begin: Retrieving Pixel Data from Camera to be Stored to
    SDRAM/////////////////

        vsync_fedge: begin
            if (vsync_1 == 0 && vsync_2 == 1)
                state_d = byte1; //vsync falling edge means new frame is
                    incoming
        end

        byte1: begin
            if (pclk_1 == 1 && pclk_2 == 0 && href_1 == 1 && href_2 == 1)
                begin
                //rising edge of pclk means new pixel data(first byte of
                    16-bit pixel RGB565) is available at output
                pixel_d[15:8] = cmos_db;
                state_d = byte2;
            end else if (vsync_1 == 1 && vsync_2 == 1) begin
                state_d = vsync_fedge;
                wr_ptr_real_d = 0;
                wr_ptr_d = 0;
            end
        end
        byte2: begin
            if (pclk_1 == 1 && pclk_2 == 0 && href_1 == 1 && href_2 == 1)
```

```
                begin
                //rising edge of pclk means new pixel data(second byte of
                    16-bit pixel RGB565) is available at output
                pixel_d[7:0] = cmos_db;
                state_d = fifo_write;
            end else if (vsync_1 == 1 && vsync_2 == 1)
                state_d = vsync_fedge;
        end

        fifo_write: begin
            //write the 16-bit data to asynchronous fifo to be retrieved
                later by SDRAM
            wr_ptr_d = wr_ptr_q + 1;
            wr_ptr_real_d = wr_ptr_real_q;
            if (wr_ptr_q < 42) begin
                if (wr_ptr_real_d != 1763) begin
                    wr_ptr_real_d = wr_ptr_real_q + 1;
                    wr_en = 1;
                    if (full)
                        led_d=4'b1001;
                end
            end else if (wr_ptr_q == 79)
                wr_ptr_d = 0;

            state_d = byte1;
        end
        default: state_d = idle;
    endcase
  end
end

assign cmos_pwdn = 0;
assign cmos_rst_n = 1;
assign led = led_q;

//module instantiations
i2c_top #(.freq(100_000)) m0
    (
    .clk(clk_100),
    .rst_n(rst_n),
    .start(start),
    .stop(stop),
    .wr_data(wr_data),
    .rd_tick(rd_tick), //ticks when read data from servant is ready,data will
        be taken from rd_data
    .ack(ack), //ack[1] ticks at the ack bit[9th bit],ack[0] asserts when ack
```

```
        bit is ACK,else NACK
    .rd_data(rd_data),
    .scl(cmos_scl),
    .sda(cmos_sda),
    .state(state)
);

clock_12_0002 clock_12_inst
(
    .refclk (clk), // refclk.clk
    .rst (rst), // reset.reset
    .outclk_0 (cmos_xclk), // outclk0.clk
    .locked () // (terminated)
);

debounce_explicit m4
(
    .clk(clk_100),
    .rst_n(rst_n),
    .sw({!key[0]}),
    .db_level(),
    .db_tick(key0_tick)
);
    //////////////////////////////////////////////////////////////////

always @(posedge clk_vga, negedge rst_n) begin
    if (!rst_n) begin
        rd_ptr_q <= 0;
    end else begin
        rd_ptr_q <= rd_ptr_d;
    end
end

always @(posedge clk_100, negedge rst_n)begin
    if (!rst_n) begin
        wr_ptr_q <= 0;
        wr_ptr_real_q <= 0;
    end else begin
        wr_ptr_q <= wr_ptr_d;
        wr_ptr_real_q <= wr_ptr_real_d;
    end
end

always @* begin
    rd_ptr_d = rd_ptr_q;
    if (rd_en_vga) begin
```

```
        if (rd_ptr_q >= 1763)
            rd_ptr_d = 0;
        else
            rd_ptr_d = rd_ptr_q + 1;
    end
end

always @* begin
    delay_storing_d = delay_storing_q;
    start_storing_d = start_storing_q;
    button_count_d = button_count_q;
    if (key0_tick)
        delay_storing_d = 1;
    if (delay_storing_q && wr_ptr_real_d == 0) begin
        start_storing_d = 1;
        delay_storing_d = 0;
    end
    if(start_storing_q && wr_ptr_real_d == 1763) begin
        start_storing_d = 0;
        button_count_d = button_count_q + 1;
    end
end

assign flag2 = start_storing_d;

//row and column counter
reg flag_for_counter;

always @(posedge clk)begin
    if(!rst_n)begin
        flag_for_counter <= 0;
        addr_avalon<=0;
    end
end

dual_port_sync_mod d0(
    .clk_r(clk_vga),
    .clk_w(clk),
    .we(wr_en),
    .rd(rd_en),
    .rd_en_to_avalon_mem(rd_en_to_avalon),
    .din(pixel_q[15:8]),
    .addr_a(wr_ptr_real_d),
    .addr_b(rd_ptr_d),
    .addr_avalon_mem(rd_ptr_from_avalon),
    .dout(dout_vga),
```

```
        .flag(flag),
        .start(start_storing_d),
        .pixel_data_mem(pixel_data)
    );

endmodule

module dual_port_sync_mod
    #(
    parameter ADDR_WIDTH=13, DATA_WIDTH=8
    )
    (
    input clk_r,
    input clk_w,
    input we,
    input rd,
    input rd_en_to_avalon_mem,
    input start,
    input[DATA_WIDTH-1:0] din,
    input[ADDR_WIDTH-1:0] addr_a,addr_b,addr_avalon_mem,
    //addr_a for write, addr_b for read
    output[DATA_WIDTH-1:0] dout,
    output reg[DATA_WIDTH-1:0] pixel_data_mem,
    output wire flag
    );

    reg[DATA_WIDTH-1:0] pixel_data;
    reg[DATA_WIDTH-1:0] ram [8191:0];
    reg[ADDR_WIDTH-1:0] addr_b_q;


    always @(posedge clk_w) begin
        if (we && start) begin
            ram[addr_a] <= din;
            flag <= 1;
        end
        else flag <= 0;
    end

    //read to vga
    always @(posedge clk_r) begin
        addr_b_q <= addr_b;
    end

    assign pixel_data_mem = ram[addr_avalon_mem];
    assign dout = ram[addr_b_q];
```

```
endmodule
```

6.3 Accelerator: Computation Kernel

```
module kernel
    #(
        parameter data_width = 8,
        parameter weight_width = 8
    ) (
        input logic    clk,
        input logic    en,
        input logic    init,
        input logic [ 1 : 0] mode,
        input logic bias_en,
        input logic [data_width-1 : 0] data,
        input logic [weight_width-1 : 0] weight,
        output logic [data_width-1 : 0] result
    );

    logic [2*data_width-1 : 0] mac_result;
    logic [data_width+data_width/2 : 0] mac_result_q;
    logic [data_width-1 : 0] relu_result;
    logic [data_width-1 : 0] comp_result;
    logic overflow, carry, checkbit;
    logic [1:0] tmp;

    always @(posedge clk) begin
        if (init) begin
            mac_result <= 0;
        end else begin
            if (en) begin
                if (bias_en == 1'b1) begin
                    mac_result_q <= mac_result[15: 4] + {{4{weight[7]}}, weight};
                    tmp[0] <= mac_result[15] ^ weight[7];
                    tmp[1] <= mac_result[15];
                end else
                    mac_result <= mac_result + {{8{data[7]}}, data} *
                        {{8{weight[7]}}, weight};
            end
        end
    end

    assign overflow = tmp[0] ? 0 : (mac_result_q[11] ^ tmp[1]);
    assign checkbit = overflow ? mac_result_q[12] : mac_result_q[11];

    assign relu_result = overflow ? (mac_result_q[12] ? 0 : mac_result_q[10:3])
        : (mac_result_q[11] ? 0 : mac_result_q[10:3]);
```

```
    always @(posedge clk) begin
        if (init) begin
            comp_result <= 0;
        end else begin
            if (en) begin
                comp_result <= (data > comp_result) ? data : comp_result;
            end
        end
    end

    assign result = (mode != 1) ? relu_result : comp_result;

endmodule
```

6.4 Accelerator: Memory Block

```
module memory
    #(
        parameter data_width  = 8,
        parameter weight_width  = 8
    ) (
        input logic clk,
        input logic rstn,

        input logic mode,
        input logic [31 : 0] data_idx,
        input logic [31 : 0] weight_idx,
        input logic [31 : 0] result_idx,
        input logic write_enable,
        input logic [ data_width-1 : 0] result,
        output logic [ data_width-1 : 0] data,
        output logic [weight_width-1 : 0] weight
    );

    logic [ data_width-1 : 0] data_a0, data_a1;

    logic [ data_width-1 : 0] memory_A0 [8191 : 0];
    logic [ data_width-1 : 0] memory_A1 [8191 : 0];
    logic [weight_width-1 : 0] memory_B [3438-1 : 0] =
        '{8'b00000000,8'b11110100,8'b00101010,8'b00100101,8'b11101101,8'b00001101,8'b00000010,8'b
        incomplete. refer to GitHub*/

    always_ff @(posedge clk) begin
        weight <= memory_B[weight_idx];
    end

    always_ff @(posedge clk) begin
        if ((mode != 1) && write_enable)
            memory_A1[result_idx] <= result;
        data_a1 <= memory_A1[data_idx];
    end

    always_ff @(posedge clk) begin
        if ((mode == 1) && write_enable)
            memory_A0[result_idx] <= result;
        data_a0 <= memory_A0[data_idx];
    end

   assign data = (mode == 1) ? data_a1 : data_a0;
```

```
endmodule
```

6.5 Accelerator: Control Unit

```
typedef struct {
    logic [31 : 0] data_idx_tmp; // Starting Address for DATA
    logic [31 : 0] weight_idx_tmp; // Starting Address for WEIGHT
    logic [31 : 0] result_idx_tmp; // Starting Address for RESULT

    logic [ 8 : 0] counter0; // Kernel Offset (0, 1, 2, 42, 43, 44, 84, 85, 86)
    logic [ 3 : 0] counter1; // Output image counter (used for weight addressing)
    logic [12 : 0] counter2; // Result Index (0 - 6400)
    logic [ 8 : 0] counter3; // Convolution Iteration (0-8)
    logic [ 2 : 0] counter4; // Input image counter (input map number - used for
        data addressing in Conv2)

    logic [12 : 0] data_offset; // Kernel Starting Position
    logic [11 : 0] weight_offset;
    logic [10 : 0] input_offset_data;
    logic [ 3 : 0] input_offset_weight;
    logic [ 5 : 0] column;  // Column Counter
    logic [ 5 : 0] row;  // Row Counter

    logic flag_out_image;
    logic flag_out_index;  // Used to increase output index
    logic bias_data;
    logic k_en;  // Enable the Kernel
    logic k_init;  // Reset the Kernel
    logic [ 1 : 0] k_mode;  // Kernel Mode (Convolution or Max Pooling)

    logic wen;  // Memory Write Enable
    logic done;  // Layer Completed
} reg_type;

module cu
    (
    input logic  clk,
    input logic  rstn,
    // Software
    input logic go,
    input logic [ 2 : 0] layer_index,
    input logic [31 : 0] data_address,
    input logic [31 : 0] data_size,
    input logic [31 : 0] weight_address,
    input logic [31 : 0] weight_size,
    input logic [31 : 0] result_address,
    output logic [ 2 : 0] done,
```

```
// Kernel
output logic en,
output logic  init,
output logic [ 1 : 0] mode, // Both Kernel and Memories
output logic bias_en,
// Memories
output logic [31 : 0] data_idx,
output logic [31 : 0] weight_idx,
output logic [31 : 0] result_idx,
output logic write_enable

);

//logic [31 : 0] weight_idx_tmp;

logic [ 4 : 0] current_state, next_state;

reg_type cs, ns;
logic [ 5 : 0] data_counter;
logic [ 3 : 0] weight_counter;

parameter S0 = 0;

// Layer 0: Conv
parameter S1 = 1;
parameter S2 = 2;
parameter S3 = 3;
parameter S4 = 4;
parameter S5 = 5;
// Layer 1: Max
parameter S6 = 6;
parameter S7 = 7;
parameter S8 = 8;
// Layer 2: Conv
parameter S9 = 9;
parameter S10 = 10;
parameter S11 = 11;
parameter S12 = 12;
parameter S13 = 13;
// Layer 3: Max
parameter S14 = 14;
parameter S15 = 15;
parameter S16 = 16;
// Layer 4: Dense
parameter S17 = 17;
parameter S18 = 18;
```

```
parameter S19 = 19;
parameter S20 = 20;
parameter S21 = 21;

always @(*) begin

    ns = cs;

    case (current_state)
        S0: begin
            ns.wen = 0;
            ns.done = cs.done;
            ns.k_init = 1'b0;
            ns.counter0 = 0;
            ns.counter1 = 0;
            ns.counter2 = 0;
            ns.counter3 = 0;
            ns.weight_offset = 0;
            ns.data_offset = 0;
            next_state = S0;

            if (go == 1'b1) begin
                ns.data_idx_tmp = data_address;
                ns.weight_idx_tmp = weight_address;
                ns.result_idx_tmp = result_address;
                ns.k_init = 1'b1;
                if (layer_index == 3'b000) begin
                    next_state = S1;
                    ns.k_mode = 2'b00;
                end
                else if (layer_index == 3'b001) begin
                    next_state = S6;
                    ns.k_mode = 2'b01;
                end
                else if (layer_index == 3'b010) begin
                    next_state = S9;
                    ns.k_mode = 2'b10;
                end
                else if (layer_index == 3'b011) begin
                    next_state = S14;
                    ns.k_mode = 2'b01;
                end
                else if (layer_index == 3'b100) begin
                    next_state = S17;
                    ns.k_mode = 2'b10;
                end
```

```
    end
end

S1: begin
    ns.counter0 = cs.counter0 + 1;    // In-Kernel Offset
    ns.counter3 = cs.counter3 + 1;    // Convolution iteration
    ns.weight_offset = cs.weight_offset + 4;
    ns.wen = 1'b0;
    next_state = S1;
    if (cs.counter3 == 0) begin // Address 0: Enable MAC
        ns.k_init = 1'b0;
        ns.k_en = 1'b1;
        if (cs.flag_out_index == 1'b1) begin // If not the first ever
            iteration: Increase output index
            ns.counter2 = cs.counter2 + 1;
            ns.flag_out_index = 1'b0;
        end
    end
    else if (cs.counter3 == 2 || cs.counter3 == 5) // Next row of
        kernel
        ns.counter0 = cs.counter0 + 40;
    else if (cs.counter3 == 8) begin
        next_state = S2;
        ns.counter0 = 0;    // Reset Iteration Counter
        ns.counter3 = 0;    // Reset Convolution Kernel
        if (cs.column != 39) begin // Move to next column (Update Data
            Offset + Column Counter)
            ns.data_offset = cs.data_offset + 1;
            ns.column = cs.column + 1;
        end else begin    // Move to next row
            ns.column = 0;
            if (cs.row != 39) begin
                ns.data_offset = cs.data_offset + 3; // Update Data
                    Offset (+3)
                ns.row = cs.row + 1;    // Update Row
            end else begin    // Move to next output map
                ns.data_offset = 0; // Reset Everything
                ns.row = 0;
                if (cs.counter1 != 3) begin
                    ns.flag_out_image = 1'b1;
                end else    // Layer END
                    next_state = S4;
                end
            end
        end
    end
```

```
    S2: begin    // Bias Address Generated
    if (cs.flag_out_image == 1'b1) begin  // If output map finished
        update the weight index
        ns.counter1 = cs.counter1 + 1;
        ns.flag_out_image = 1'b0;
    end
    ns.weight_offset = 0;
    ns.bias_data = 1'b1;
    next_state = S3;
end

S3: begin    // Initialize MAC for next Convolution
    ns.bias_data = 1'b0;
    ns.k_init = 1'b1;
    ns.k_en = 1'b0;
    ns.wen = 1'b1;    // Write Result to Memory
    ns.flag_out_index = 1'b1;
    next_state = S1;
end

S4: begin
    ns.k_en = 0;
    ns.bias_data = 1'b1;
    next_state = S5;
end

S5: begin
    ns.bias_data = 1'b0;
    ns.wen = 1;
    ns.done = cs.done + 1;
    next_state = S0;
end

S6: begin
    next_state = S6;
    ns.counter0 = cs.counter0 + 1;    // In-Kernel Offset
    ns.counter3 = cs.counter3 + 1;    // Convolution iteration
    ns.wen = 1;
    if (cs.counter3 == 0) begin  // Address 0: Enable MAC
        ns.k_init = 1'b0;
        ns.k_en = 1'b1;
        if (cs.flag_out_index == 1'b1) begin // If not the first ever
            iteration: Increase output index
            ns.counter2 = cs.counter2 + 1;
            ns.flag_out_index = 1'b0;
```

```
        end
    end
    else if (cs.counter3 == 1)  // Next row of kernel
        ns.counter0 = cs.counter0 + 39;
    else if (cs.counter3 == 3) begin
        next_state = S7;
        ns.counter0 = 0;    // Reset Iteration Counter
        ns.counter3 = 0;    // Reset Convolution Kernel

        if (cs.column != 38) begin // Move to next column (Update Data
            Offset + Column Counter)
            ns.data_offset = cs.data_offset + 2;
            ns.column = cs.column + 2;
        end else begin   // Move to next row
            ns.column = 0;
            ns.data_offset = cs.data_offset + 42;
            if (cs.row != 38) begin
                // Update Data Offset (+3)
                ns.row = cs.row + 2;  // Update Row
            end else begin   // Move to next output map
                ns.row = 0;
                if (cs.counter1 != 3) begin
                    ns.counter1 = cs.counter1 + 1;
                end else begin  // Layer END
                    next_state = S8;
                    ns.data_offset = 0;
                end
            end
        end
    end
end

S7: begin  // Initialize Kernel for next MAX Pooling
    ns.k_init = 1'b1;
    ns.k_en = 1'b0;
    ns.wen = 1'b1;    // Write Result to Memory
    ns.flag_out_index = 1'b1;
    next_state = S6;
end

S8: begin
    ns.k_init = 1'b1;
    ns.k_en = 1'b0;
    ns.wen = 1;
    ns.done = cs.done + 1;
    next_state = S0;
```

```
end

S9: begin
    next_state = S9;
    ns.counter0 = cs.counter0 + 1;    // In-Kernel Offset
    ns.counter3 = cs.counter3 + 1;    // Convolution iteration
    ns.weight_offset = cs.weight_offset + 16;
    ns.wen = 1'b0;

    if (cs.counter3 == 0) begin // Address 0: Enable MAC
        ns.k_init = 1'b0;
        ns.k_en = 1'b1;
        if (cs.flag_out_index == 1'b1) begin // If not the first ever
            iteration: Increase output index
            ns.counter2 = cs.counter2 + 1;
            ns.flag_out_index = 1'b0;
        end
    end

    else if (cs.counter3 == 2 || cs.counter3 == 5) // Next row of
        kernel
        ns.counter0 = cs.counter0 + 18;

    else if (cs.counter3 == 8) begin
        ns.counter0 = 0;     // Reset Iteration Counte
        ns.counter3 = 0;     // Reset Convolution Kernel

        if (cs.counter4 != 3) begin
            ns.counter4 = cs.counter4 + 1;
            ns.input_offset_data = cs.input_offset_data + 400;
            ns.input_offset_weight = cs.input_offset_weight + 4;
            ns.weight_offset = 0;
        end else begin
            ns.counter4 = 0;
            ns.input_offset_data = 0;
            ns.input_offset_weight = 0;
            next_state = S10;
            if (cs.column != 17) begin // Move to next column (Update
                Data Offset + Column Counter)
                ns.data_offset = cs.data_offset + 1;
                ns.column = cs.column + 1;
            end else begin    // Move to next row
                ns.column = 0;
                if (cs.row != 17) begin
                    ns.data_offset = cs.data_offset + 3; // Update Data
                        Offset (+3)
```

```
                ns.row = cs.row + 1;   // Update Row
            end else begin   // Move to next output map
                ns.data_offset = 0; // Reset Everything
                ns.row = 0;
                if (cs.counter1 != 3) begin
                    ns.flag_out_image = 1'b1;
                end else   // Layer END
                    next_state = S12;
            end
        end
    end
end
end

S10: begin    // Bias Address Generated
    if (cs.flag_out_image == 1'b1) begin  // If output map finished
        update the weight index
        ns.counter1 = cs.counter1 + 1;
        ns.flag_out_image = 1'b0;
    end
    ns.weight_offset = 0;
    ns.bias_data = 1'b1;
    next_state = S11;
end

S11: begin    // Initialize MAC for next Convolutio
    ns.bias_data = 1'b0;
    ns.k_init = 1'b1;
    ns.k_en = 1'b0;
    ns.wen = 1'b1;    // Write Result to Memor
    ns.flag_out_index = 1'b1;
    next_state = S9;
end

S12: begin
    ns.k_en = 0;
    ns.bias_data = 1'b1;
    next_state = S13;
end

S13: begin
    ns.bias_data = 1'b0;
    ns.wen = 1;
    ns.done = cs.done + 1;
    next_state = S0;
end
```

```
S14: begin
    next_state = S14;
    ns.counter0 = cs.counter0 + 1;    // In-Kernel Offset
    ns.counter3 = cs.counter3 + 1;    // Convolution iteration
    ns.wen = 1'b0;

    if (cs.counter3 == 0) begin  // Address 0: Enable MAC
        ns.k_init = 1'b0;
        ns.k_en = 1'b1;
        if (cs.flag_out_index == 1'b1) begin // If not the first ever
            iteration: Increase output index
            ns.counter2 = cs.counter2 + 1;
            ns.flag_out_index = 1'b0;
        end
    end

    else if (cs.counter3 == 1)  // Next row of kernel
        ns.counter0 = cs.counter0 + 17;

    else if (cs.counter3 == 3) begin
        next_state = S15;
        ns.counter0 = 0;    // Reset Iteration Counter
        ns.counter3 = 0;    // Reset Convolution Kernel

        if (cs.column != 16) begin // Move to next column (Update Data
            Offset + Column Counter)
            ns.data_offset = cs.data_offset + 2;
            ns.column = cs.column + 2;
        end else begin   // Move to next row
            ns.column = 0;
            ns.data_offset = cs.data_offset + 20;
            if (cs.row != 16) begin
                ns.row = cs.row + 2;  // Update Row
            end else begin   // Move to next output map
                ns.row = 0;
                if (cs.counter1 != 3) begin
                    ns.counter1 = cs.counter1 + 1;
                end else begin  // Layer END
                    next_state = S16;
                end
            end
        end
    end
end
```

```
S15: begin    // Initialize Kernel for next MAX Pooling
    ns.k_init = 1'b1;
    ns.k_en = 1'b0;
    ns.wen = 1'b1;    // Write Result to Memory
    ns.flag_out_index = 1'b1;
    next_state = S14;
end

S16: begin
    ns.k_init = 1'b1;
    ns.k_en = 1'b0;
    ns.wen = 1;
    ns.done = cs.done + 1;
    next_state = S0;
end

S17: begin
          next_state = S17;
          ns.counter0 = cs.counter0 + 1;    // In-Kernel Offset
          ns.counter3 = cs.counter3 + 1;    // Convolution iteration
          ns.weight_offset = cs.weight_offset + 10;
          ns.wen = 1'b0;
          if (cs.counter3 == 0) begin // Address 0: Enable MAC
        ns.k_init = 1'b0;
        ns.k_en = 1'b1;
        if (cs.flag_out_index == 1'b1) begin // If not the first ever
            iteration: Increase output index
            ns.counter2 = cs.counter2 + 1;
            ns.flag_out_index = 1'b0;
        end
    end
    else if (cs.counter3 == 323) begin
        next_state = S18;
        ns.counter0 = 0;    // Reset Iteration Counter
        ns.counter3 = 0;    // Reset Convolution Kernel
        if (cs.counter1 != 9) begin
            ns.flag_out_image = 1'b1;
        end else    // Layer END
            next_state = S20;
    end
end

S18: begin    // Bias Address Generated
    if (cs.flag_out_image == 1'b1) begin  // If output map finished
        update the weight index
        ns.counter1 = cs.counter1 + 1;
```

```
            ns.flag_out_image = 1'b0;
        end
        ns.weight_offset = 0;
        ns.bias_data = 1'b1;
        next_state = S19;
    end

    S19: begin     // Initialize MAC for next Convolutio
        ns.bias_data = 1'b0;
        ns.k_init = 1'b1;
        ns.k_en = 1'b0;
        ns.wen = 1'b1;    // Write Result to Memor
        ns.flag_out_index = 1'b1;
        next_state = S17;
    end

    S20: begin
        ns.k_en = 0;
        ns.bias_data = 1'b1;
        next_state = S21;
    end

    S21: begin
        ns.bias_data = 1'b0;
        ns.wen = 1;
        ns.done = 0;
        next_state = S0;
    end
  endcase
end

assign data_idx = cs.data_idx_tmp + cs.counter0 + cs.data_offset +
    cs.input_offset_data;
assign weight_idx = cs.weight_idx_tmp + cs.counter1 + cs.weight_offset +
    cs.input_offset_weight;
assign result_idx = cs.result_idx_tmp + cs.counter2;
assign en = cs.k_en;
assign init = cs.k_init;
assign mode = cs.k_mode;
assign write_enable = cs.wen;
assign bias_en = cs.bias_data;
assign done = cs.done;


always @(posedge clk) begin
  if (!rstn) begin
```

```
            current_state <= S0;
            cs.k_init <= 1'b0;
            cs.k_en <= 1'b0;
            cs.k_mode <= 0;
            cs.done <= 0;
            cs.wen <= 1'b0;
            cs.counter0 <= 0;
            cs.counter1 <= 0;
            cs.counter2 <= 0;
            cs.counter3 <= 0;
            cs.counter4 <= 0;
            cs.data_offset <= 0;
            cs.weight_offset <= 0;
            cs.input_offset_data <= 0;
            cs.input_offset_weight <= 0;
            cs.column <= 0;
            cs.row <= 0;
            cs.flag_out_image <= 0;
            cs.flag_out_index <= 0;
            cs.bias_data <= 0;
        end else begin
            current_state <= next_state;
            cs <= ns;
        end
    end
endmodule
```

6.6 Accelerator: Top

```
    module top
    #(
        parameter data_width  = 8,
        parameter weight_width  = 8
    ) (
        input logic clk,
        input logic rstn,

        input logic [data_width-1 : 0] image,
        output logic [31 : 0] image_idx,

        // Software
        input logic go,
        input logic [ 2 : 0] layer_index,
        input logic [31 : 0] data_address,
        input logic [31 : 0] data_size,
        input logic [31 : 0] weight_address,
        input logic [31 : 0] weight_size,
        input logic [31 : 0] result_address,
        output logic [ 2 : 0] done
);

    // Control Unit -> Kernel
    logic en;
    logic  init;
    logic [1:0] mode;
    logic bias_en;

    // Control Unit -> Memories
    logic [31 : 0] data_idx;
    logic [31 : 0] weight_idx;
    logic [31 : 0] result_idx;
    logic write_enable;

    // Memories -> Kernel
    logic [ data_width-1 : 0] result;
    logic [ data_width-1 : 0] data;
    logic [weight_width-1 : 0] weight;
    logic [ data_width-1 : 0] data_kernel;

    assign image_idx = data_idx;
    assign data_kernel = (mode == 0) ? image : data;

    cu control_unit
```

```
(
    .clk(clk),
    .rstn(rstn),
    .go(go),
    .layer_index(layer_index),
    .data_address( data_address),
    .data_size(data_size),
    .weight_address(weight_address),
    .weight_size(weight_size),
    .result_address(result_address),
    .done(done),
    .en(en),
    .init(init),
    .mode(mode),
    .bias_en(bias_en),
    .data_idx(data_idx),
    .weight_idx(weight_idx),
    .result_idx(result_idx),
    .write_enable(write_enable)
);

kernel  kernel_unit
(
    .clk(clk),
    .en(en),
    .init(init),
    .mode(mode),
    .bias_en(bias_en),
    .data(data_kernel),
    .weight(weight),
    .result(result)
);

memory memory_unit
(
    .clk(clk),
    .rstn(rstn),
    .mode(mode),
    .data_idx(data_idx),
    .weight_idx(weight_idx),
    .result_idx(result_idx),
    .write_enable(write_enable),
    .data(data),
    .weight(weight),
    .result(result)
);
```

```
endmodule
```

6.7 Accelerator: RTL Testbench

```
module testbench();

    reg clk;
    reg rstn;
    reg go;
    reg [ 2 : 0] layer_index;
    reg [31 : 0] data_address;
    reg [31 : 0] data_size;
    reg [31 : 0] weight_address;
    reg [31 : 0] weight_size;
    reg [31 : 0] result_address;
    wire  done;

    reg [31 : 0] result_size;
    top top_0 (.*);

    integer i;
    integer fd_conv1_out, fd_max1_out, fd_conv2_out, fd_max2_out, fd_dense_out;

    always begin
        `HALF_CLOCK_PERIOD;
        clk = ~clk;
    end

    initial begin

        $readmemb("binary_weights.txt", `WEIGHT_MEM);
        $readmemb("binary_image.txt", `DATA_MEMO);

        // register setup
        clk = 0;
        rstn = 0;
        layer_index = 0;
        data_address = 0;
        data_size = 1764;
        weight_address = 0;
        weight_size = 40;
        result_address = 0;
        result_size = 6400;

        @(posedge clk);

        @(negedge clk); // release resetn
        rstn = 1;
```

```
go = 1;

@(posedge clk);
go = 0;
@(posedge clk); // start the first cycle

while (done != 1) begin
   @(posedge clk); // next cycle
end
@(posedge clk);

fd_conv1_out = $fopen("rtl_out_conv1.txt", "w");

for (i = 0; i < result_size; i = i + 1) begin
   $fdisplayb(fd_conv1_out, top_0.memory_unit.memory_A1[i]);
end

@(posedge clk);
@(posedge clk);

/////////////////
// MAX POOLING //
/////////////////

layer_index = 1;
data_address = 0;
data_size = 6400;
weight_address = 0;
weight_size = 0;
result_address = 0;
result_size = 1600;

@(posedge clk);
@(negedge clk); // release resetn
go = 1;

@(posedge clk);
go = 0;
@(posedge clk); // start the first cycle

while (done != 1) begin
   @(posedge clk); // next cycle
end
@(posedge clk);

fd_max1_out = $fopen("rtl_out_max1.txt", "w");
```

```
for (i = 0; i < result_size; i = i + 1) begin
    $fdisplayb(fd_max1_out, top_0.memory_unit.memory_A0[i]);
end

@(posedge clk);
@(posedge clk);


///////////////////
/// CONV LAYER 2 //
///////////////////

layer_index = 2;
data_address = 0;
data_size = 1600;
weight_address = 40;
weight_size = 148;
result_address = 0;
result_size = 1296;

@(posedge clk);
@(negedge clk); // release resetn
go = 1;

@(posedge clk);
go = 0;
@(posedge clk); // start the first cycle

while (done != 1) begin
    @(posedge clk); // next cycle
end
@(posedge clk);

fd_conv2_out = $fopen("rtl_out_conv2.txt", "w");
for (i = 0; i < result_size; i = i + 1) begin
    $fdisplayb(fd_conv2_out, top_0.memory_unit.memory_A1[i]);
end

@(posedge clk);
@(posedge clk);

///////////////////
// MAX POOLING 2 //
///////////////////
```

```
layer_index = 3;
data_address = 0;
data_size = 1296;
weight_address = 0;
weight_size = 0;
result_address = 0;
result_size = 324;

@(posedge clk);
@(negedge clk); // release resetn
go = 1;

@(posedge clk);
go = 0;
@(posedge clk); // start the first cycle

while (done != 1) begin
   @(posedge clk); // next cycle
end
@(posedge clk);

fd_max2_out = $fopen("rtl_out_max2.txt", "w");

for (i = 0; i < result_size; i = i + 1) begin
   $fdisplayb(fd_max2_out, top_0.memory_unit.memory_A0[i]);
end

@(posedge clk);
@(posedge clk);

///////////////
/// DENSE ///
///////////////

layer_index = 4;
data_address = 0;
data_size = 324;
weight_address = 188;
weight_size = 3250;
result_address = 0;
result_size = 10;

@(posedge clk);
@(negedge clk); // release resetn
go = 1;
```

```
        @(posedge clk);
        go = 0;
        @(posedge clk); // start the first cycle

        while (done != 1) begin
            @(posedge clk); // next cycle
        end
        @(posedge clk);

        fd_dense_out = $fopen("rtl_out_dense.txt", "w");

        for (i = 0; i < result_size; i = i + 1) begin
            $fdisplayb(fd_dense_out, top_0.memory_unit.memory_A1[i]);
        end

        @(posedge clk);
        @(posedge clk);

        $finish;
    end

endmodule // testbench
```

6.8 Verilator Testbench

```
#include <iostream>
#include "Vtop.h"
#include <verilated.h>
#include <verilated_vcd_c.h>
#include <fstream>
#include <string>

int main(int argc, const char ** argv, const char ** env) {

        Verilated::commandArgs(argc, argv);
        int exitcode = 0;

        Vtop * dut = new Vtop; // Instantiate the top module

        // Enable dumping a VCD file

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


        std::ifstream image_file, weight_file;
        image_file.open("binary_image.txt");
        weight_file.open("binary_weights.txt");

        std::string read_value;
        uint8_t int_value;
        char print_value;

        std::ofstream fd_conv1, fd_max1, fd_conv2, fd_max2, fd_dense;
        fd_conv1.open("ver_out_conv1.txt");
        fd_max1.open("ver_out_max1.txt");
        fd_conv2.open("ver_out_conv2.txt");
        fd_max2.open("ver_out_max2.txt");
        fd_dense.open("ver_out_dense.txt");

        std::ifstream fd_conv1_gold, fd_max1_gold, fd_conv2_gold, fd_max2_gold,
            fd_dense_gold;
        fd_conv1_gold.open("/user/stud/fall23/vp2518/EmbeddedSystems/Project_v3/MNIST_HW_AXRL-mai
        fd_max1_gold.open("/user/stud/fall23/vp2518/EmbeddedSystems/Project_v3/MNIST_HW_AXRL-main
        fd_conv2_gold.open("/user/stud/fall23/vp2518/EmbeddedSystems/Project_v3/MNIST_HW_AXRL-mai
        fd_max2_gold.open("/user/stud/fall23/vp2518/EmbeddedSystems/Project_v3/MNIST_HW_AXRL-main
        fd_dense_gold.open("/user/stud/fall23/vp2518/EmbeddedSystems/Project_v3/MNIST_HW_AXRL-mai
```

```
int identical = 10;


int counter = 0;
if (image_file.is_open()) {
	while (std::getline(image_file, read_value)) {
		int_value = std::stoi(read_value, nullptr, 2);
		dut->top__DOT__memory_unit__DOT__memory_A0[counter] =
		    int_value;
		counter++;
	}
	image_file.close();
}

counter = 0;
if (weight_file.is_open()) {
	while (std::getline(weight_file, read_value)) {
		int_value = std::stoi(read_value, nullptr, 2);
		dut->top__DOT__memory_unit__DOT__memory_B[counter] =
		    int_value;
		counter++;
	}
	weight_file.close();
}


unsigned int data_addresses[5] = {0, 0, 0, 0, 0};
unsigned int data_sizes[5] = {1764, 6400, 1600, 1296, 324};
unsigned int weight_addresses[5] = {0, 0, 40, 0, 188};
unsigned int weight_sizes[5] = {40, 0, 148, 0, 3250};
unsigned int result_addresses[5] = {0, 0, 0, 0, 0};
unsigned int result_sizes[5] = {6400, 1600, 1296, 324, 10};

uint8_t conv1_gold[result_sizes[0]];
uint8_t max1_gold[result_sizes[1]];
uint8_t conv2_gold[result_sizes[2]];
uint8_t max2_gold[result_sizes[3]];
uint8_t dense_gold[result_sizes[4]];

counter = 0;
if (fd_conv1_gold.is_open()) {
	while (std::getline(fd_conv1_gold, read_value)) {
		int_value = std::stoi(read_value, nullptr, 2);
		conv1_gold[counter] = int_value;
```

```
                counter++;
        }
        fd_conv1_gold.close();
}

counter = 0;
if (fd_max1_gold.is_open()) {
        while (std::getline(fd_max1_gold, read_value)) {
                int_value = std::stoi(read_value, nullptr, 2);
                max1_gold[counter] = int_value;
                counter++;
        }
        fd_max1_gold.close();
}

counter = 0;
if (fd_conv2_gold.is_open()) {
        while (std::getline(fd_conv2_gold, read_value)) {
                int_value = std::stoi(read_value, nullptr, 2);
                conv2_gold[counter] = int_value;
                counter++;
        }
        fd_conv2_gold.close();
}

counter = 0;
if (fd_max2_gold.is_open()) {
        while (std::getline(fd_max2_gold, read_value)) {
                int_value = std::stoi(read_value, nullptr, 2);
                max2_gold[counter] = int_value;
                counter++;
        }
        fd_max2_gold.close();
}

counter = 0;
if (fd_dense_gold.is_open()) {
        while (std::getline(fd_dense_gold, read_value)) {
                int_value = std::stoi(read_value, nullptr, 2);
                dense_gold[counter] = int_value;
                counter++;
        }
        fd_dense_gold.close();
}

bool last_clk = true;
```

```
int time = 0;
int time_limit;
int time_go = 0;

// Initial values

dut->go = 0;
dut->rstn = 0;

int hist_done = 0;

for (int k = 0 ; k < 8 ; k++, time += 10) {
        dut->clk = ((time % 20) >= 10) ? 1 : 0;
        if (time == 20) dut->rstn = 1; // Pulse "go" for 1 cycles
        dut->eval();
        tfp->dump(time);
}

for (int i = 0 ; i < 5 ; i++) {

        time_limit = time + 20000000;
        time_go = time;

        std::cout << "Layer " << i << " Start" << std::endl;

        dut->layer_index = i;
        dut->data_address = data_addresses[i];
        dut->data_size = data_sizes[i];
        dut->weight_address = weight_addresses[i];
        dut->weight_size = weight_sizes[i];
        dut->result_address = result_addresses[i];

        for (;time < time_limit ; time += 10) {
                dut->clk = ((time % 20) >= 10) ? 1 : 0; // Simulate a 50
                    MHz clock
                if (time == time_go + 20) dut->go = 1; // Pulse "go" for 1
                    cycles
                if (time == time_go + 40) dut->go = 0;

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


                if (dut->clk && !last_clk && !dut->go) {
                        if (dut->done != hist_done) {
                                hist_done = dut->done;
```

```
                    break;  // Stop once "done" appears
                }
            }
            last_clk = dut->clk;
        }


        // Once "done" is received, run a few more clock cycles

        for (int k = 0 ; k < 4 ; k++, time += 10) {
            dut->clk = ((time % 20) >= 10) ? 1 : 0;
            dut->eval();
            tfp->dump(time);
        }

        for (int k = 0; k < result_sizes[i]; k++) {

            if (i % 2 == 0)
                int_value =
                    dut->top__DOT__memory_unit__DOT__memory_A1[k];
            else
                int_value =
                    dut->top__DOT__memory_unit__DOT__memory_A0[k];



            for (int l = 7; l >= 0; l--) {
                print_value = (int_value & (1 << l)) ? '1' : '0';

                if  (i == 0) {
                    fd_conv1 << print_value;
                    if (int_value != conv1_gold[k]) {
                        identical = 0;
                    }
                }
                else if (i == 1) {
                    fd_max1 << print_value;
                    if (int_value != max1_gold[k]) identical = 1;
                }
                else if (i == 2) {
                    fd_conv2 << print_value;
                    if (int_value != conv2_gold[k]) identical =
                        2;
                }
                else if (i == 3) {
                    fd_max2 << print_value;
```

```
                        if (int_value != max2_gold[k]) identical = 3;
                    }
                    else if (i == 4) {
                        fd_dense << print_value;
                        if (int_value != dense_gold[k]) identical =
                            4;
                    }
                }
                if  (i == 0) fd_conv1 << std::endl;
                else if (i == 1) fd_max1 << std::endl;
                else if (i == 2) fd_conv2 << std::endl;
                else if (i == 3) fd_max2 << std::endl;
                else if (i == 4) fd_dense << std::endl;
            }

            if (identical == 10)
                std::cout << "Layer " << i << " Done: Correct" <<
                    std::endl;
            else {
                std::cout << "Layer " << identical << " Done: False" <<
                    std::endl;
                return(1);
            }

            std::cout << std::endl;
        }

        std::cout << "RTL Model " << " Done: Correct" << std::endl;

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

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

        return 0;
}
```

6.9 Driver C File

```
/* * Device driver for the VGA video generator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 * drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_ball.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_ball.c
 */

#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 "vga_ball.h"

#define DRIVER_NAME "vga_ball"

/* Device registers */
#define BG_RED(x) (x)
#define BG_GREEN(x) ((x)+1)
#define BG_BLUE(x) ((x)+2)
#define H(x) ((x)+3)
#define V(x) ((x)+5)
```

```
/*
 * Information about our device
 */
struct vga_ball_dev {
        struct resource res; /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
        vga_ball_color_t background;
        vga_ball_hv_t hv;
        vga_ball_read_t read_val;
        vga_ball_read_pixel_t read_pixel;
        vga_ball_write_pixel_t write_pixel;
        vga_ball_write_pixel_2_t write_pixel_2;
} dev;
/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up */
static short read_val(void)
{
        unsigned char incoming = ioread8(BG_RED(dev.virtbase));
        return incoming;
}

static char read_pixel(void)
{
        unsigned char incoming_pixel = ioread8(BG_GREEN(dev.virtbase));
        return incoming_pixel;
}

static void write_pixel(vga_ball_write_pixel_t *write_pixel)
{
        iowrite8(write_pixel->pixel_addr_lower, BG_RED(dev.virtbase) );
        dev.write_pixel = *write_pixel;
}

static void write_pixel_2(vga_ball_write_pixel_2_t *write_pixel_2)
{
        iowrite8(write_pixel_2->pixel_addr_upper, BG_GREEN(dev.virtbase) );
        dev.write_pixel_2 = *write_pixel_2;
}


static void write_background(vga_ball_color_t *background)
{
        iowrite8(background->red, BG_RED(dev.virtbase) );
        iowrite8(background->green, BG_GREEN(dev.virtbase) );
        iowrite8(background->blue, BG_BLUE(dev.virtbase) );
```

```
        dev.background = *background;
}

static void write_hv(vga_ball_hv_t *hv){
        iowrite16(hv->h, H(dev.virtbase));
        iowrite16(hv->v, V(dev.virtbase));
        dev.hv = *hv;
}
/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
        vga_ball_arg_t vla;

        switch (cmd) {
        case VGA_BALL_WRITE_BACKGROUND:
                if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                                   sizeof(vga_ball_arg_t)))
                        return -EACCES;
                write_background(&vla.background);
                break;

        case VGA_BALL_READ_BACKGROUND:
                vla.background = dev.background;
                if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                                 sizeof(vga_ball_arg_t)))
                        return -EACCES;
                break;
        case VGA_BALL_WRITE_HV:
                if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                                   sizeof(vga_ball_arg_t)))
                        return -EACCES;
                write_hv(&vla.hv);
                break;

        case VGA_BALL_READ_HV:
                vla.hv = dev.hv;
                if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                                 sizeof(vga_ball_arg_t)))
                        return -EACCES;
                break;

        case VGA_BALL_READ_VAL:
```

```
                vla.read_val.val = read_val();

                if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                                 sizeof(vga_ball_arg_t)))
                        return -EACCES;

                break;
         case VGA_BALL_READ_PIXEL:
                vla.read_pixel.pixel_val = read_pixel();

                if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                                 sizeof(vga_ball_arg_t)))
                        return -EACCES;

                break;
         case VGA_BALL_WRITE_PIXEL:
                if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                                   sizeof(vga_ball_arg_t)))
                        return -EACCES;
                write_pixel(&vla.write_pixel);
                break;

        case VGA_BALL_WRITE_PIXEL_2:
                if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                                   sizeof(vga_ball_arg_t)))
                        return -EACCES;
                write_pixel_2(&vla.write_pixel_2);
                break;


        default:
                return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_ball_fops = {
        .owner  = THIS_MODULE,
        .unlocked_ioctl = vga_ball_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_ball_misc_device = {
        .minor  = MISC_DYNAMIC_MINOR,
```

```
        .name   = DRIVER_NAME,
        .fops   = &vga_ball_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_ball_probe(struct platform_device *pdev)
{
        vga_ball_color_t beige = { 0xf9, 0xe4, 0xb7 };
        vga_ball_hv_t initial  = {0x6, 0x6};
        int ret;

        /* Register ourselves as a misc device: creates /dev/vga_ball */
        ret = misc_register(&vga_ball_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

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

        return 0;

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

```
/* Clean-up code: release resources */
static int vga_ball_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&vga_ball_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_ball_of_match[] = {
	{ .compatible = "csee4840,vga_ball-1.0" },
	{},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_ball_driver = {
	.driver = {
		.name = DRIVER_NAME,
		.owner = THIS_MODULE,
		.of_match_table = of_match_ptr(vga_ball_of_match),
	},
	.remove = __exit_p(vga_ball_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_ball_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit vga_ball_exit(void)
{
	platform_driver_unregister(&vga_ball_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_ball_init);
module_exit(vga_ball_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA ball driver");
```

6.10 DRIVER H FILE

```
#ifndef _VGA_BALL_H
#define _VGA_BALL_H

#include <linux/ioctl.h>

typedef struct{
	unsigned short val;
} vga_ball_read_t;

typedef struct{
	unsigned char pixel_val;
} vga_ball_read_pixel_t;

typedef struct{
	unsigned char pixel_addr_lower;
	//unsigned char pixel_addr_upper;
} vga_ball_write_pixel_t;

typedef struct{
	unsigned char pixel_addr_upper;
	//unsigned char pixel_addr_upper;
} vga_ball_write_pixel_2_t;

typedef struct {
	unsigned char red, green, blue;
} vga_ball_color_t;

typedef struct {
	unsigned short h, v;
} vga_ball_hv_t;

typedef struct {
  vga_ball_color_t background;
  vga_ball_hv_t hv;
  vga_ball_read_t read_val;
  vga_ball_read_pixel_t read_pixel;
  vga_ball_write_pixel_t write_pixel;
  vga_ball_write_pixel_2_t write_pixel_2;
} vga_ball_arg_t;
```

```
#define VGA_BALL_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_BALL_WRITE_BACKGROUND _IOW(VGA_BALL_MAGIC, 1, vga_ball_arg_t *)
#define VGA_BALL_READ_BACKGROUND _IOR(VGA_BALL_MAGIC, 2, vga_ball_arg_t *)
#define VGA_BALL_WRITE_HV _IOW(VGA_BALL_MAGIC, 3, vga_ball_arg_t *)
#define VGA_BALL_READ_HV _IOR(VGA_BALL_MAGIC, 4, vga_ball_arg_t *)
#define VGA_BALL_READ_VAL _IOR(VGA_BALL_MAGIC, 5,vga_ball_arg_t *)
#define VGA_BALL_READ_PIXEL _IOR(VGA_BALL_MAGIC, 6,vga_ball_arg_t *)
#define VGA_BALL_WRITE_PIXEL _IOW(VGA_BALL_MAGIC, 7,vga_ball_arg_t *)
#define VGA_BALL_WRITE_PIXEL_2 _IOW(VGA_BALL_MAGIC, 8,vga_ball_arg_t *)
#endif
\end{verbatim}

\subsection{C Software}
\begin{verbatim}

#include <stdio.h>
#include "vga_ball.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdbool.h>
#include <stdint.h>
int vga_ball_fd;

/* Read and print the background color */
void print_background_color() {
  vga_ball_arg_t vla;

  if (ioctl(vga_ball_fd, VGA_BALL_READ_BACKGROUND, &vla)) {
      perror("ioctl(VGA_BALL_READ_BACKGROUND) failed");
      return;
  }
  printf("%02x %02x %02x\n",
        vla.background.red, vla.background.green, vla.background.blue);
}

/* Set the background color */
void set_background_color(const vga_ball_color_t *c)
{
  vga_ball_arg_t vla;
  vla.background = *c;
```

```
      if (ioctl(vga_ball_fd, VGA_BALL_WRITE_BACKGROUND, &vla)) {
      perror("ioctl(VGA_BALL_SET_BACKGROUND) failed");
      return;
  }
}

void set_hv(const vga_ball_hv_t *c)
{
  vga_ball_arg_t vla;
  vla.hv = *c;
  if (ioctl(vga_ball_fd, VGA_BALL_WRITE_HV, &vla)) {
      perror("ioctl(VGA_BALL_SET_HV) failed");
      return;
  }
}

static unsigned char read_val()
{
 vga_ball_arg_t vla;

 if (ioctl(vga_ball_fd, VGA_BALL_READ_VAL, &vla)) {
      perror("ioctl(VGA_BALL_READ) failed");
      return;
  }
 return vla.read_val.val;
}

static unsigned char read_pixel()
{
 vga_ball_arg_t vla;

 if (ioctl(vga_ball_fd, VGA_BALL_READ_PIXEL, &vla)) {
      perror("ioctl(VGA_BALL_READ) failed");
      return;
  }
 return vla.read_pixel.pixel_val;
}

static unsigned char write_pixel(const vga_ball_write_pixel_t *c)
{
  vga_ball_arg_t vla;
  vla.write_pixel = *c;
      if (ioctl(vga_ball_fd, VGA_BALL_WRITE_PIXEL, &vla)) {
      perror("ioctl(VGA_BALL_SET_PIXEL_ADDR) failed");
      return;
  }
```

```
}

static unsigned char write_pixel_2(const vga_ball_write_pixel_2_t *c)
{
  vga_ball_arg_t vla;
  vla.write_pixel_2 = *c;
      if (ioctl(vga_ball_fd, VGA_BALL_WRITE_PIXEL_2, &vla)) {
      perror("ioctl(VGA_BALL_SET_PIXEL_ADDR) failed");
      return;
  }
}


int main()
{
  vga_ball_arg_t vla;
  int i;
  static const char filename[] = "/dev/vga_ball";

  static const vga_ball_color_t colors[] = {
    { 0xff, 0x00, 0x00 }, /* Red */
    { 0x00, 0xff, 0x00 }, /* Green */
    { 0x00, 0x00, 0xff }, /* Blue */
    { 0xff, 0xff, 0x00 }, /* Yellow */
    { 0x00, 0xff, 0xff }, /* Cyan */
    { 0xff, 0x00, 0xff }, /* Magenta */
    { 0x80, 0x80, 0x80 }, /* Gray */
    { 0x00, 0x00, 0x00 }, /* Black */
    { 0xff, 0xff, 0xff } /* White */
  };

  vga_ball_hv_t hv_val = {0, 0};

# define COLORS 9

  printf("VGA ball Userspace program started\n");

  if ( (vga_ball_fd = open(filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", filename);
    return -1;
  }

  printf("initial state: ");

  unsigned char incoming;
  unsigned char incoming_old = -1;
```

```
unsigned char incoming_pixel;
unsigned char addr_lower=0;
unsigned char addr_upper=0;
vga_ball_write_pixel_t write_val;
vga_ball_write_pixel_2_t write_val_2;
uint16_t  counter;
int old_done=0;
int done;
int8_t results[10];

while(1) {
     incoming = read_val();

     if(incoming != incoming_old){

            printf("Button press: %d\n", incoming);
            incoming_old = incoming;
            write_val_2.pixel_addr_upper = 0;
            write_pixel_2(&write_val_2);

            for (int i = 0; i < 5; i++) {

                   write_val.pixel_addr_lower = 0x08 | i;
                   write_pixel(&write_val);
                   printf("Layer %d Go\n", i);
                   printf("Value given: %d\n", 0x08 | i);
                   write_val.pixel_addr_lower = 0;
                   write_pixel(&write_val);
                   printf("Go deasserted\n");

                   while((done = read_pixel())==old_done);
                   old_done = done;
                   printf("Layer %d Done\n", i);

            }

            printf("Accelerator completed\n");
            write_val_2.pixel_addr_upper = 0x80;

            for(int i = 0; i < 10; i++){
            write_pixel_2(&write_val_2);
            write_val_2.pixel_addr_upper++;
            results[i] = read_pixel();
            printf("Digit %d: %d\n", i, results[i]);
            }
     }
```

```
  }

  printf("VGA BALL Userspace program terminating\n");
  return 0;
}
```

6.11 C-Simulator

```
void HW_FP(int8_t *weights, int weights_address, int weights_size, int8_t *data,
    int data_address, int data_size, int layer_index, int8_t *out)
{
    if (layer_index == 0) // Convolution & ReLU
    {
        int out_index = 0;

        for (int i = 0 ; i < 4 ; i++)
        {
            int count = 0;
            for (int j = 0 ; j < 42 * 42; j++) //output size will be 40x40
            {
                if ((j - 42*count) / 42 == 1) count++;

                if ((((j % (42*count + 40) != 0) && (j % (42*count + 41) != 0))
                    || (j == 0)) && (count < 40))
                {
                    int16_t mac_res = 0b0000000000000000;

                    mac_res += data[data_address + j ] * weights[weights_address +
                        i ];
                    mac_res += data[data_address + j + 1] *
                        weights[weights_address + i + 4];
                    mac_res += data[data_address + j + 2] *
                        weights[weights_address + i + 8];
                    mac_res += data[data_address + j + 42] *
                        weights[weights_address + i + 12];
                    mac_res += data[data_address + j + 43] *
                        weights[weights_address + i + 16];
                    mac_res += data[data_address + j + 44] *
                        weights[weights_address + i + 20];
                    mac_res += data[data_address + j + 84] *
                        weights[weights_address + i + 24];
                    mac_res += data[data_address + j + 85] *
                        weights[weights_address + i + 28];
                    mac_res += data[data_address + j + 86] *
                        weights[weights_address + i + 32];

                    mac_res = (mac_res >> 4); // Shift right by 4
                    mac_res += weights[weights_address + i + 36]; //Bias

                    mac_res = mac_res>>3;
```

```
                out[out_index] = (mac_res > 0x0000) ? mac_res :
                    0b0000000000000000;

                out_index++;
            }

        }
    }
}

else if (layer_index == 1) // Max Pool 1
{
    int out_index = 0;

    for (int i = 0; i < 4; i++) //Now here the image is the out of the
        previous layer
    {
        for (int j = 0; j < 40 * 20; j += 2)
        {
            int8_t max;

            max = data[data_address + i * 40 * 40 + (j % 40) + (j - (j % 40))
                * 2 ];
            max = (data[data_address + i * 40 * 40 + (j % 40) + (j - (j %
                40)) * 2 + 1] > max) ? data[data_address + i * 40 * 40 + (j %
                40) + (j - (j % 40)) * 2 + 1] : max;
            max = (data[data_address + i * 40 * 40 + (j % 40) + (j - (j %
                40)) * 2 + 40] > max) ? data[data_address + i * 40 * 40 + (j %
                40) + (j - (j % 40)) * 2 + 40] : max;
            max = (data[data_address + i * 40 * 40 + (j % 40) + (j - (j %
                40)) * 2 + 41] > max) ? data[data_address + i * 40 * 40 + (j %
                40) + (j - (j % 40)) * 2 + 41] : max;

            out[out_index] = max;
            out_index++;
        }
    }
}

else if (layer_index == 2) //Conv 2 and ReLU
{
    int out_index = 0;

    for (int i = 0; i < 4; i++)
    {
        int count = 0;
```

```
        for (int j = 0; j < 20 * 20; j++)
        {
            if ((j - 20*count) / 20 == 1) count++;

            if ((((j % (20*count + 18) != 0) && (j % (20*count + 19) != 0))
                || (j == 0)) && (count < 18))
            {
                int16_t mac_res = 0b0000000000000000;

                for (int k = 0; k < 4; k++)
                {

                    mac_res += data[data_address + j + k * 20 * 20] *
                        weights[weights_address + i + (k * 4) ];
                    mac_res += data[data_address + j + 1 + k * 20 * 20] *
                        weights[weights_address + i + (k * 4) + 16];
                    mac_res += data[data_address + j + 2 + k * 20 * 20] *
                        weights[weights_address + i + (k * 4) + 32];
                    mac_res += data[data_address + j + 20 + k * 20 * 20] *
                        weights[weights_address + i + (k * 4) + 48];
                    mac_res += data[data_address + j + 21 + k * 20 * 20] *
                        weights[weights_address + i + (k * 4) + 64];
                    mac_res += data[data_address + j + 22 + k * 20 * 20] *
                        weights[weights_address + i + (k * 4) + 80];
                    mac_res += data[data_address + j + 40 + k * 20 * 20] *
                        weights[weights_address + i + (k * 4) + 96];
                    mac_res += data[data_address + j + 41 + k * 20 * 20] *
                        weights[weights_address + i + (k * 4) + 112];
                    mac_res += data[data_address + j + 42 + k * 20 * 20] *
                        weights[weights_address + i + (k * 4) + 128];

                }
                mac_res = (mac_res >> 4);

                mac_res += weights[weights_address + i + 144];

                mac_res = mac_res >> 3; //Divide by 8

                out[out_index] = (mac_res > 0x0000) ? mac_res :
                    0b0000000000000000;
                out_index++;
            }
        }
    }
}
```

```
else if (layer_index == 3) // Max Pool 2
{
    int out_index = 0;

    for (int i = 0; i < 4; i++) //Now here the image is the out of the
        previous layer
    {
        for (int j = 0; j < 18 * 9; j += 2)
        {
            int8_t max;

            max = data[data_address + i * 18 * 18 + (j % 18) + (j - (j % 18))
                * 2 ];
            max = (data[data_address + i * 18 * 18 + (j % 18) + (j - (j %
                18)) * 2 + 1] > max) ? data[data_address + i * 18 * 18 + (j %
                18) + (j - (j % 18)) * 2 + 1] : max;
            max = (data[data_address + i * 18 * 18 + (j % 18) + (j - (j %
                18)) * 2 + 18] > max) ? data[data_address + i * 18 * 18 + (j %
                18) + (j - (j % 18)) * 2 + 18] : max;
            max = (data[data_address + i * 18 * 18 + (j % 18) + (j - (j %
                18)) * 2 + 19] > max) ? data[data_address + i * 18 * 18 + (j %
                18) + (j - (j % 18)) * 2 + 19] : max;

            out[out_index] = max;
            out_index++;
        }
    }
}

else if (layer_index == 4) // Dense Layer
{

    for (int i = 0; i < 10; i++)
    {
        int16_t mac_res = 0b0000000000000000;

            for (int k = 0; k < 9 * 9 * 4; k++)
            {
                mac_res += data[data_address + k] * weights[weights_address +
                    (k * 10) + i];
            }

        mac_res = (mac_res >> 4); // Shift right by 4

        mac_res += weights[weights_address + 3240 + i];
```

```
            mac_res = mac_res >> 3;

            out[i] = (mac_res > 0x0000) ? mac_res : 0b0000000000000000;

        }
    }
}

//Converter function: for converting the floating point weights and bias into
    our fixed point representation

// -8 4 2 1 . 0.5 0.25 0.125 0.0625

void converter(float num)
{

        int int_part = (int) num;

        float frac_part = num - int_part;

        frac_part = fabs(frac_part);

        if((num < 0 && num > -1) || (num < -1 && num > -2) || (num < -2 && num >
            -3) || (num < -3 && num > -4) || (num < -4 && num > -5) || (num < -5
            && num > -6) || (num < -7 && num > -8))
            frac_part = 1 - frac_part;

        printf("0b"); //C format for binary

        if(num < 0)
        {
            printf("1");
            num = num + 8;
            if(num >= 4)
            {
                printf("1");
                num = num - 4;
            }
            else
                printf("0");

            if(num >= 2)
            {
                printf("1");
                num = num - 2;
            }
```

```
    else
        printf("0");

    if(num >= 1)
    {
        printf("1");
        num = num - 1;
    }
    else
        printf("0");
}

else if(num >= 0)
{
    printf("0");

    if(num >= 4)
    {
        printf("1");
        num = num - 4;
    }
    else
        printf("0");

    if(num >= 2)
    {
        printf("1");
        num = num - 2;
    }
    else
        printf("0");

    if(num >= 1)
    {
        printf("1");
        num = num - 1;
    }
    else
        printf("0");
}

if(frac_part >= 0.5)
{
    printf("1");
    frac_part = frac_part - 0.5;
}
```

```
        else
            printf("0");

        if(frac_part >= 0.25)
        {
            printf("1");
            frac_part = frac_part - 0.25;
        }
        else
            printf("0");

        if(frac_part >= 0.125)
        {
            printf("1");
            frac_part = frac_part - 0.125;
        }
        else
            printf("0");

        if(frac_part >= 0.0625)
        {
            printf("1");
            frac_part = frac_part - 0.0625;
        }
        else
            printf("0");

    // }
}
```

6.12 Model Training

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
import tensorflow as tf
from tensorflow.keras.models import load_model
from tensorflow.keras import Model
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.utils import to_categorical
from skimage.transform import resize
import h5py
import os
import cv2

# Resize as 42x42(Camera like), make it between 0 and 127(Divide each pixel by
    2), so that it can be represented in 7 bits

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train_resized = np.zeros((x_train.shape[0], 42, 42), dtype=np.uint8)
for i in range(x_train.shape[0]):
    x_train_resized[i] = cv2.resize(x_train[i], (42, 42))

x_test_resized = np.zeros((x_test.shape[0], 42, 42), dtype=np.uint8)
for i in range(x_test.shape[0]):
    x_test_resized[i] = cv2.resize(x_test[i], (42, 42))

x_train_resized = x_train_resized.reshape(-1, 42, 42, 1)
x_test_resized = x_test_resized.reshape(-1, 42, 42, 1)

# Divide by 2
for i in range(x_train.shape[0]):
    x_train_resized[i] = x_train_resized[i]/2

for i in range(x_test.shape[0]):
    x_test_resized[i] = x_test_resized[i]/2

x_test_max = np.max(x_test_resized)
print(x_test_max)
print(x_test_resized.shape)

# One-hot encode labels
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

```
# Convert into bits, and then convert again in Floating point but using
# Our Fixed point Representation: [-8, 4, 2, 1, 0.5, 0.25, 0.125, 0.0625] So,
    127 -> 7.95

def convert_to_binary(images):
    binary_images = []

    for img in images:
        binary_img = []

        for pixel_value in np.nditer(img):
            binary_str = format(pixel_value, '08b')
            binary_img.append(binary_str)

        binary_images.append(binary_img)

    return binary_images

# Convert x_test_resized and x_train_resized to binary
binary_x_test_resized = convert_to_binary(x_test_resized)
binary_x_train_resized = convert_to_binary(x_train_resized)

def binary_to_float(binary_str):
    weights = [-8, 4, 2, 1, 0.5, 0.25, 0.125, 0.0625]

    result = 0

    for i, bit in enumerate(binary_str):
        result += int(bit) * weights[i]

    return result

def convert_to_float(binary_images):
    float_images = []

    for img in binary_images:
        float_img = []

        for binary_str in img:
            float_value = binary_to_float(binary_str)
            float_img.append(float_value)

        float_img = np.array(float_img).reshape((42, 42))
        float_images.append(float_img)
```

```
    return np.array(float_images)

# Convert binary_x_test_resized and binary_x_train_resized to floating point
    values as numpy arrays
FP_x_test_resized = convert_to_float(binary_x_test_resized)
FP_x_train_resized = convert_to_float(binary_x_train_resized)

FP_x_train_resized = FP_x_train_resized.reshape(-1, 42, 42, 1)
FP_x_test_resized = FP_x_test_resized.reshape(-1, 42, 42, 1)

## Training of the model.....

# import numpy as np
# from keras.models import Sequential
# from keras.layers import Conv2D, MaxPooling2D, Flatten, Dropout, Dense
# from keras.callbacks import ModelCheckpoint

# model = Sequential()
# model.add(Conv2D(4, kernel_size=(3, 3), activation='relu', input_shape=(42,
    42, 1)))
# model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Conv2D(4, kernel_size=(3, 3), activation='relu'))
# model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Flatten())
# model.add(Dropout(0.5))
# model.add(Dense(10, activation='softmax'))

# # Compile the model
# model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])

# model.fit(FP_x_train_resized, y_train, batch_size=50, epochs=15,
    validation_data=(FP_x_test_resized, y_test))

# model.save('model_weights_FP.h5')

# Rearrange Weights for our dense layer (Format in Python is not practical for
    hardware)

import numpy as np
text_file = np.loadtxt("model_weights_FP.txt")
# print(text_file[188:3428])

array_b = np.zeros(3240)
array_a = text_file[188:3428]
print(array_a)
```

```
for i in range(10):
    for j in range(4):
        for k in range(9*9):
            array_b[i + k * 10 + (j * 9 * 9 * 10)] = array_a[(j * 10) + (k * 40)
                + i]

array_final = np.zeros(3438)

array_final[0:188] = text_file[0:188]
array_final[188:3428] = array_b
array_final[3428:3438] = text_file[3428:3438]


with open('model_weights_FP_rearrange.txt', 'w') as file:
    for layer_weights in array_final:
            file.write(str(layer_weights) + '\n')

print("Model weights saved to model_weights.txt")


np.savetxt("model_weights_FP_rearrange.txt", array_final, delimiter=',',
    fmt='%f')

# Scaling the weights from -8 to 8: Better in terms of Accuracy and for Fixed
    Point Model

import numpy as np
float_values = np.loadtxt("model_weights_FP_rearrange.txt")

max_value = np.max(np.abs(float_values))

scaled_values = (float_values/max_value) * 8

np.savetxt("model_weights_FP_scaled.txt", scaled_values, fmt='%f')
```