DESIGN DOCUMENT FOR FPGA NEURAL NETWORK ACCELERATION

Stephen Ogunmwonyi (iso2107)

Aymen Ahmed Norain (aan2161)

Bradley Chen Jocelyn (bcj2124)

Connor James Espenshade (cje2136)

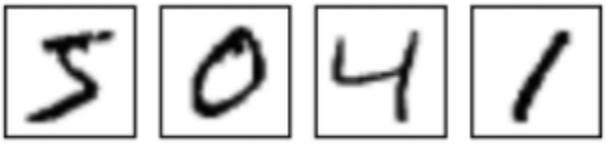
Table of Contents:

Introduction

This project focused on establishing the foundational software interface for an FPGA-accelerated neural network designed for MNIST digit recognition. The implemented system consists of a userspace application (`user.c`) running on the Altera DE10-SoC's HPS, which communicates with a conceptual hardware accelerator on the FPGA fabric via a custom Linux kernel driver (`control.c`). The core achievement from the software perspective is the successful implementation and testing of the communication protocol. This includes the userspace logic for loading bit-packed image data (pixel, address, and a load completion signal) to the hardware, polling a hardware status register for computation completion, and retrieving the resulting 10-element dense layer output (classification scores) from the accelerator.

Problem Identification & Dataset

For this project, we sought to classify a series of images as representing certain digits. That is, given a black and white, low-resolution image of a handwritten digit, the model implemented in our hardware/software interface running directly on the FPGA should classify which number the digit most closely resembles. To solve this problem, we selected the MNIST dataset, a collection of 70,000 grayscale images of handwritten digits. Each digit is 28x28 pixels, with each pixel stored as an 8-bit grayscale value ranging from 0 to 255, where 0 is black and 255 is white.



To reduce memory complexity, we binarize each pixel: black pixels are represented with a 0 and white pixels represented with a 1. This allows us to store each pixel in 1 bit, or each image in 784 bits, or 98 bytes. Understanding the train/test split in the dataset—where testing images are not used to determine the weights during training—to be 60,000 and 10,000 images for model training and testing, all 10,000 test images would fit in just under 1MB. For this proof-of-concept stage, however, we will consider 1 image.

Model Architecture

We carefully evaluated multiple algorithms to implement on the device. We considered implementing large general CNNs for general image classification: Resnet-18, SqueezeNet, and MobileNet. While these large CNNs do not require any additional training, they require a great amount of memory and have their own quirks including residual connections and complex interconnected webs of convolutions. As we are developing models to detect MNIST data samples, this is a much simpler problem space and does not require implementing a full CNN off-the-shelf. Instead, we intend to construct our own neural networks from Keras and TensorFlow tutorials, one leveraging convolution and one only leveraging dense layers (Chollet, "Training"). We intend to build the basis for a system that can take simple Keras models trained off device and perform inference live, allocating various hardware accelerating components to speed up different Keras layers. Therefore, convolution, pooling, padding, flattening, and dense models (matrix multiplication) should be built in hardware as much as possible.

For this accelerator, we implemented the following model in TensorFlow:

```python
model = keras.Sequential([
    keras.Input(shape=input_shape),
    layers.Conv2D(12, (3,3), activation="relu"),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(num_classes, activation="softmax"),
])
```

This model is largely inspired by a Keras tutorial, with the exception of removing the second convolution layer as well as paring down the number of filters on the first convolution

layer, adaptations that resulted in a less than 1% drop in accuracy (Chollet). The first convolution layer has 12 filters and applies convolution with a kernel size of 3x3, followed by a ReLU activation function. The results, with output shape 12x26x26, are then pooled to 12x13x13, before being flattened, where the multidimensional array is unspooled to a unidimensional vector. Finally, these 2,028 inputs are fed into a dense layer, where the pooled convolution results are condensed into the 10 output labels (digits) available from training.

Considering convolution more seriously, consider Figure 1. The 3x3 kernel selects a 3x3 window of pixels, originating in the top left corner. For some 3x3 grid, each pixel is multiply by the corresponding weight (top/middle/bottom left/middle/right, depending on location within the grid). These products are then summed to determine the output for that kernel location and filter, before sliding the filter over one column. This is repeated until the end of the row is met, where the kernel window then scans to the beginning of the row and shifts one row down until all pixels are processed. To reduce excess information, the pooling module takes a separate 2x2 window, and outputs the maximum value of the four values in the window, reducing memory 4x.

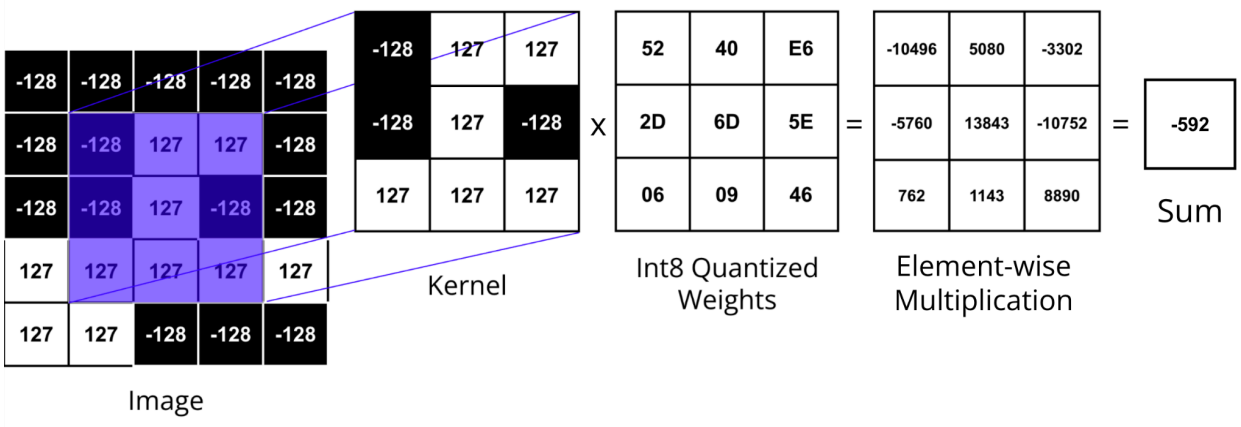


Figure 1: Convolutional 2D Computation of One Output, Quantized Weights without Scale Factor

The dense layer, on the other hand, takes the flattened 12x13x13 output from pooling and performs matrix multiplication between each output and one weight. Since there are 2028 inputs and 10 output classes, there are 20,280 weights and 10 biases, one per each possible outcome. The matrix multiplication occurs to create one element per 2028 weights. The first 2028 multiplications, summed together, correspond to the dense layer's output for the classification of a '0,' the next 2028 summed multiplications to the classification of a '1,' and so on.

Quantization

This model has over 20,410 parameters. If stored in their base format of float32, this corresponds to well over 81KB, just for the model alone. Additionally, using float32 requires difficult-to-implement floating-point arithmetic, which we hoped to avoid. For MNIST, these model weights do not require a full 32-bits of precision. There are two main data types of quantization, float16 and int8 (*Quantization*). Float 16 simply removes the extra 16-bits of precision and reduces the memory usage by 2x, but still relies on floating-point arithmetic. Int8 on the other hand, not only reduces the memory by 4x compared to float32, but also allows our model to process integer arithmetic. As such, we elected to explore quantizing the model to int8.

We quantized the MNIST model using TensorFlow Lite with Keras 3. This interpreter was verified to have the same accuracy as the full precision model at 97.87% (compared to 97.85% for the full precision). Further, we exported the weights for each layer as a series of two's complement hex numbers to be stored directly on-chip within Verilog modules.

When interacting with quantized inputs, weights, and outputs, scale factors and zero pointers are crucial. That is, when converting real value x to quantized value q, one must apply the relation $q = round(\frac{x}{S}) + z$ for scale S and zero point z. The inverse relation is $x = S \times (x_q - Z)$. These factors are present in int8 quantization, not float16 quantization, as

int8 only has 256 values to encode information. As a result, scale and zero-point factors are added to relatively adjust the impact of various features and layers across the model.

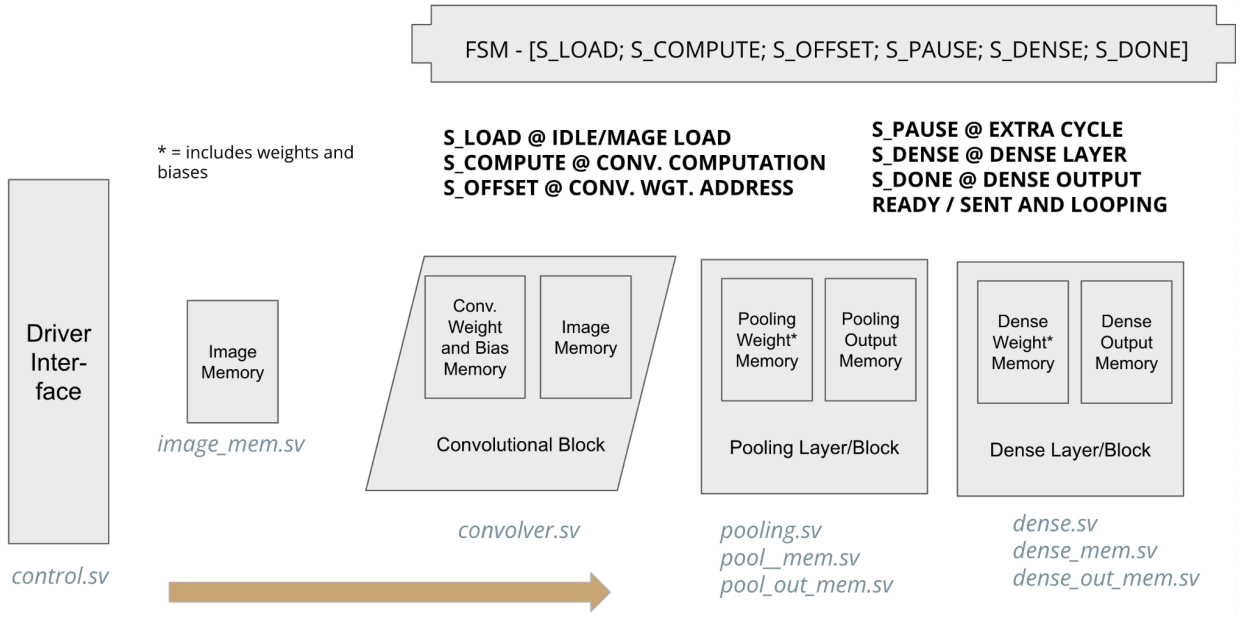
For our model, the following is a table of scale factors and zero points, applied for layer inputs, weights, and biases. Note, the zero point on the Conv2D input layer was applied as a ternary operator: with inputs 0 and 1, the zero point for 0 was simply -128, and for 1, 1/0.00392 -128 is approximately 127. This explains the -128/127 relation seen in Figure 1.

Layer	Type	Scale	Zero Point
Conv2D	Input	0.00392	-128
Conv2D	Weight	0.01033	0
Conv2D	Bias	0.000041	0
Dense	Input	0.198817	44
Dense	Weight	0.005921	0
Dense	Bias	0.000068	0
Output	Output	0.003906	-128

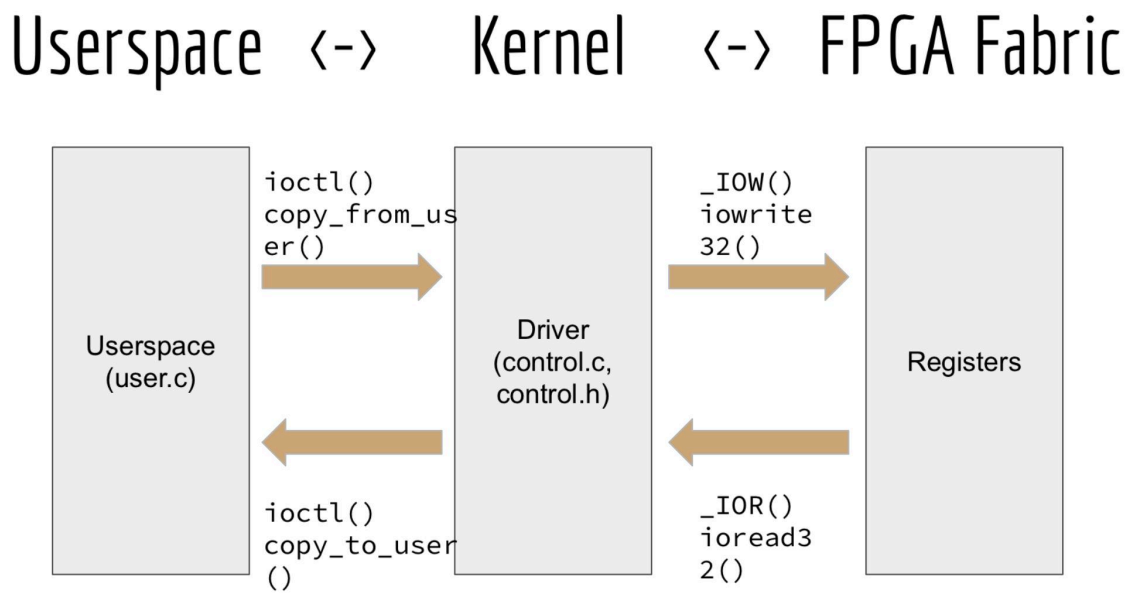
For the model, int8 weights, inputs, and biases were fed into each layer. Accumulations, done with partial multipliers and adders, were performed in int32, to ensure the model could handle large partial sums without going over. Following the conclusion of a given sum, the int32 accumulation was re-quantized back down to int8 with the scale relation $M = \frac{w \cdot x}{y}$, where the weight scale factor was multiplied by the ratio of the input and output scale factors. Because these scale factors were small, irrational decimals, they were approximated closely with bitshifts. Focusing on the Conv2D layer as an example, $M = \frac{0.01033 \times 0.00392}{0.003906} = 0.01037$. By

approximating this scale factor with a bit shift of 10 and a multiplier of 11, $2^{-10} \times 11 = 0.01074$, which is a difference of only 3.56%. This error can be absorbed by other layers, and floating point arithmetic can be completely avoided in our implementation.

General System Architecture / Associated Files Overview



Software File Definitions

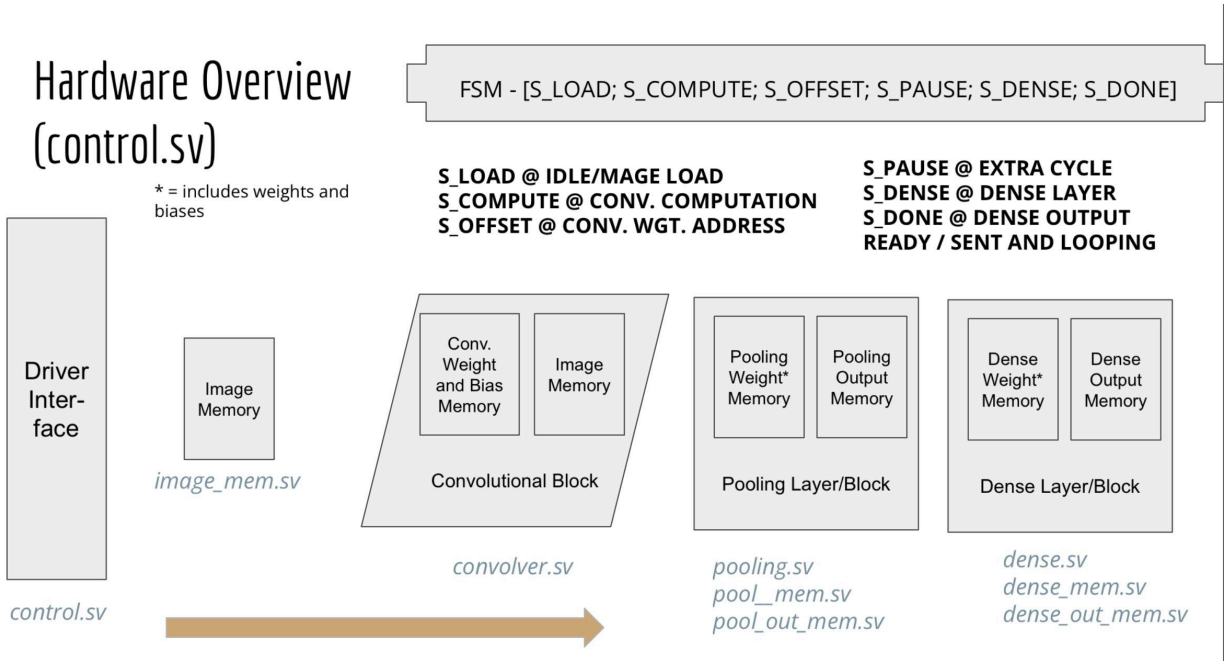


Userspace Application: user.c

Kernel Driver: control.c

Header Interface for user.c and control.c: control.h

Hardware File Definitions



Top-Level Module: top

We implemented a top level module that pretty much was an Avalon memory mapped agent. This was a very similar design to lab 3 and enabled communication using the available bits on the Avalon bus for simple and efficient communication between the software and hardware components.

Memory Modules: image_mem, data_mem, etc.

We had a series of memory modules to store key data such as the weight memory and the bias memory. We also instantiated multiple memory modules for partially computed values. We would store the values of the pooling layer in an on chip RAM whilst we waited for the 12 filters to finish convolving and then we stored the final dense layer outputs on board an on chip RAM to give the software a memory to access the data from.

Conv_weights_mem = 108 x 8

Conv_bias_mem = 10 x 8

Dense_weights_mem = 2028 x 8

Dense_weights_bias = 10 x 8

Control Unit (with FSM):

The control unit was implemented as an FSM with 6 different states to trigger different periods of operation in the model. We initially started with a pre load state where we remained till the image data was loaded and we received a signal from the software that this was complete. Following this we triggered into a compute state that ran the convolutions for the first filter set. We then had an offset state that last for a single cycle but was used to increment the weight address values so we could read in the next state. This was repeated consistently till we had

completed all 12 filters worth of convolutions. After this we were triggered into a paused state and the next state was the dense state. We had been storing the outputs of the pooling layer in an on chip RAM and we then fed this into the Dense layer to perform the massive dot product over a series of cycles. At the end of this, and once all 10 dot products were computed we shifted into a done cycle and awaited the software sending an address so we could read the final outputs out to software. All internal calculations were computed in int32 and going being layers we applied reqauntization scaling factors and zero point values

Control Unit Interface:

```
C/C++
input logic clock,
input logic reset,
    //----------- image data ----------------------
input logic image_load_done,
input logic image_data,
input logic image_write_en,
input logic [9:0] img_wr_addr,
input logic [3:0] dense_address,
output logic signed [31:0] dense_output,
output logic dense_done,

// ------------ conv ->  Pool handshake debug
output logic conv_valid_out, // valid signal to pool layer
output logic signed [31:0] mac_out, // convolution output to the pool layer
output logic pool_valid_out,
output logic signed [7:0] pooled_out,
output logic [16:0] counter_dbg
```

Convolution unit:

This unit was fed in a single pixel every clock cycle and using a rotating line buffer architecture we were able to parallelize the computation of the convolutions. In essence, we utilized a case statement to decide which row of our 4 row buffer are being written to. Each line buffer contains 28, 8-bit values that correspond to a single row. When a row is being written to, the other 3 rows

are being read from a pixel at a time. This enabled us to set up a 3x3 window and perform multiple multiplications at the same time. It also enabled We then utilized an adder tree to reduce the computations required in a single clock cycle and thus increase our Fmax. We then passed on the calculated mac_out signal to the pooling unit directly instead of storing to minimize our memory utilization.

```
C/C++
input logic clock,
input logic reset,
input logic data_valid_in,
input logic [WORD-1:0] pixel_data_in,
input logic [ADDR_W-1:0] hcount_in,
input logic signed [7:0] weight_in,
input logic weight_en,
input logic pool_valid,
input logic signed [7:0] bias,

output logic signed [31:0] mac_out,
output logic [ADDR_W-1:0] hcount,
output logic signed [7:0] mem0_out [0:HOR-1],
output logic signed [7:0] mem1_out [0:HOR-1],
output logic signed [7:0] mem2_out [0:HOR-1],
output logic signed [7:0] mem3_out [0:HOR-1],
output logic signed [7:0] top_line_out [0:2],
output logic signed [7:0] mid_line_out [0:2],
output logic signed [7:0] bot_line_out [0:2]
```

Line buffer method:

```
C/C++
valid <= data_valid_in;
        hcount_out <= hcount_in;
        if (data_valid_in) begin
            /// write incoming pixel into current memory
            case (current_write)
                2'd0 : mem0[hcount_in] <= pixel_data_in;
                2'd1 : mem1[hcount_in] <= pixel_data_in;
                2'd2 : mem2[hcount_in] <= pixel_data_in;
```

```
            2'd3 : mem3[hcount_in] <= pixel_data_in;
            default: mem0[hcount_in] <= 0;
        endcase
    end
        ///// reading values from the array
        unique case (current_write +2'd3)
            2'd0 : out0 <= mem0[hcount_in];
            2'd1 : out0 <= mem1[hcount_in];
            2'd2 : out0 <= mem2[hcount_in];
            2'd3 : out0 <= mem3[hcount_in];
            default: out0 <= 0;
        endcase

        unique case (current_write +2'd2)
            2'd0 : out1 <= mem0[hcount_in];
            2'd1 : out1 <= mem1[hcount_in];
            2'd2 : out1 <= mem2[hcount_in];
            2'd3 : out1 <= mem3[hcount_in];
            default: out1 <= 0;
        endcase

        unique case (current_write +2'd1)
            2'd0 : out2 <= mem0[hcount_in];
            2'd1 : out2 <= mem1[hcount_in];
            2'd2 : out2 <= mem2[hcount_in];
            2'd3 : out2 <= mem3[hcount_in];
            default: out2 <= 0;
        endcase

        if ((hcount_in == 27)) begin
        current_write <= current_write + 1'd1;
    end
```

Convolution Computation, Partial Adders Over 5 Cycles:

```
part_mult0 <= $signed({{24{top_line[0][7]}}, top_line[0]}) * $signed({{24{weight[0][7]}}, weight[0]});
part_mult1 <= $signed({{24{top_line[1][7]}}, top_line[1]}) * $signed({{24{weight[1][7]}}, weight[1]});
part_mult2 <= $signed({{24{top_line[2][7]}}, top_line[2]}) * $signed({{24{weight[2][7]}}, weight[2]});

part_mult3 <= $signed({{24{mid_line[0][7]}}, mid_line[0]}) * $signed({{24{weight[3][7]}}, weight[3]});
part_mult4 <= $signed({{24{mid_line[1][7]}}, mid_line[1]}) * $signed({{24{weight[4][7]}}, weight[4]});
part_mult5 <= $signed({{24{mid_line[2][7]}}, mid_line[2]}) * $signed({{24{weight[5][7]}}, weight[5]});

part_mult6 <= $signed({{24{bot_line[0][7]}}, bot_line[0]}) * $signed({{24{weight[6][7]}}, weight[6]});
part_mult7 <= $signed({{24{bot_line[1][7]}}, bot_line[1]}) * $signed({{24{weight[7][7]}}, weight[7]});
part_mult8 <= $signed({{24{bot_line[2][7]}}, bot_line[2]}) * $signed({{24{weight[8][7]}}, weight[8]});

partial1_add1 <= part_mult0 + part_mult1;
partial1_add2 <= part_mult2 + part_mult3;
partial1_add3 <= part_mult4 + part_mult5;
partial1_add4 <= part_mult6 + part_mult7;

partial2_add1 <= partial1_add1 + partial1_add2;
partial2_add2 <= partial1_add3 + partial1_add4;

partial3_add1 <= partial2_add1 + partial2_add2;

acc <= partial3_add1 + part_mult8 + $signed({{24{bias[7]}}, bias});
```

Pooling Unit:

The pooling unit was fed a single convolution output per cycle and this utilized a similar line buffer micro architecture as the convolution unit. This helped us parallelize the comparisons need to perform max pooling and also through the implementation of a comparison tree we were able to greatly increase our Fmax. We also implemented the requantization scaling factor and zero points at the end of this unit instead of the pooling unit as we were computing in int32 internally and need to output to the dense layer in int8. The scaling factor was implemented as a bit shift and multiplication and enabled us to scale the values we got back down to their correct range.

Pooling Interface:

```
input logic clock,
input logic reset,
input logic data_valid_in,
input logic signed [WORD-1:0] data_in,
input logic [ADDR_W-1:0] hcount_in,

output logic signed [7:0] pooled_out,
output logic signed [WORD-1:0] mem0_out [0:HOR-1],
output logic signed [WORD-1:0] mem1_out [0:HOR-1],
output logic signed [WORD-1:0] mem2_out [0:HOR-1],
output logic signed [WORD-1:0] mem3_out [0:HOR-1],
output logic signed [WORD-1:0] top_line_out [0:1],
output logic signed [WORD-1:0] mid_line_out [0:1]
```

Max pooling logic with scaling quantization (bit shift & multiply):

```
comparison1 <= (mid_line[1] > mid_line[0]) ? mid_line[1] : mid_line[0];
comparison2 <= (top_line[1] > top_line[0]) ? top_line[1] : top_line[0];
comparison3 <= (comparison1 > comparison2) ? (((11*comparison1) >>> 10)) : (((11*comparison2) >>> 10));
```

The Dense Unit:

This was a simple multiply accumulate module that was fed a weight and data point on each clock cycle. Again all the calculations were done in int32 internally. This took longer to compute but was a simpler microarchitecture. At the end of this calculation on cycle 2029, the scaling factors and zero point for this layer were implemented in order to scale the values back into an int8 range. This enables us to have readable values in the output and makes computation outputs more easily interpretable.

```
module dense(
    input logic clock,
    input logic reset,
    input logic signed [7:0] data_in,
    input logic signed [7:0] weight,
    input logic enable,
    input logic signed [7:0] dense_bias,

    output logic signed [31:0] output_data,
    output logic dense_valid,
    output logic [10:0] count_dbg,
    output logic [3:0] times_out
);
```

Dense Computation Logic:

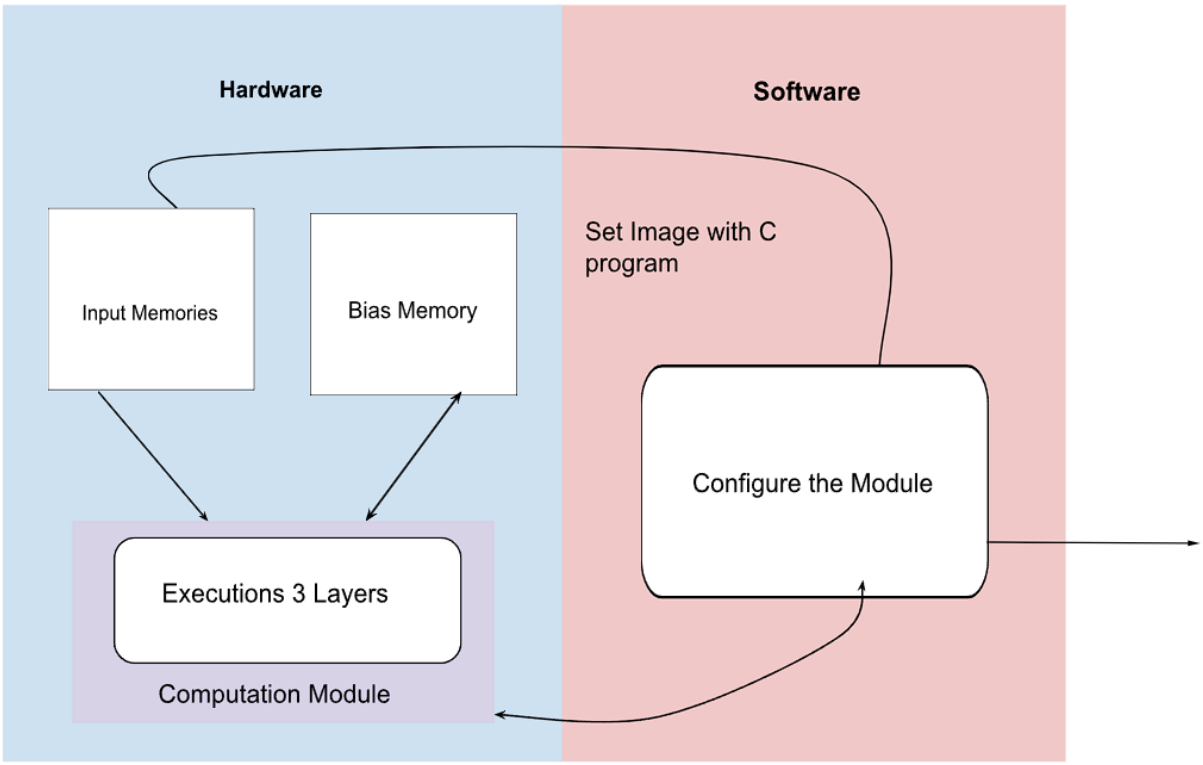
```
mult <= ($signed({{8{data_in[7]}}, data_in})) * $signed({{8{weight[7]}}, weight});
sum <=  sum + mult;
```

```
assign output_data =  ((count == 2029) && (times < 10)) ? ((11*(sum+dense_bias +44)>>>15)-128) : 0;
assign dense_valid = ((count == 2029) && (times < 10)) ? 1 : 0;
```

System Level Block Diagram

System Level Dataflow

This is a high level block diagram of the system we hope to implement. As you can see in the figure below we will have a HW/SW interface on the FPGA that will enable our hardware to communicate with our control C program on the local terminal:



As you can see above the overarching system level architecture is relatively simple to implement. The hardware side of the system consists of 3 major components. There will be N image RAMs for us to store the dataset images. As the MNIST dataset consists of 28 x 28 pixel images, the RAMs will be just under half a kilobyte in size. There will also potentially be a bias memory implemented. A bias can be defined as a constant which is added to the product of features and weights. It is used to offset the result and it helps the models to shift the activation

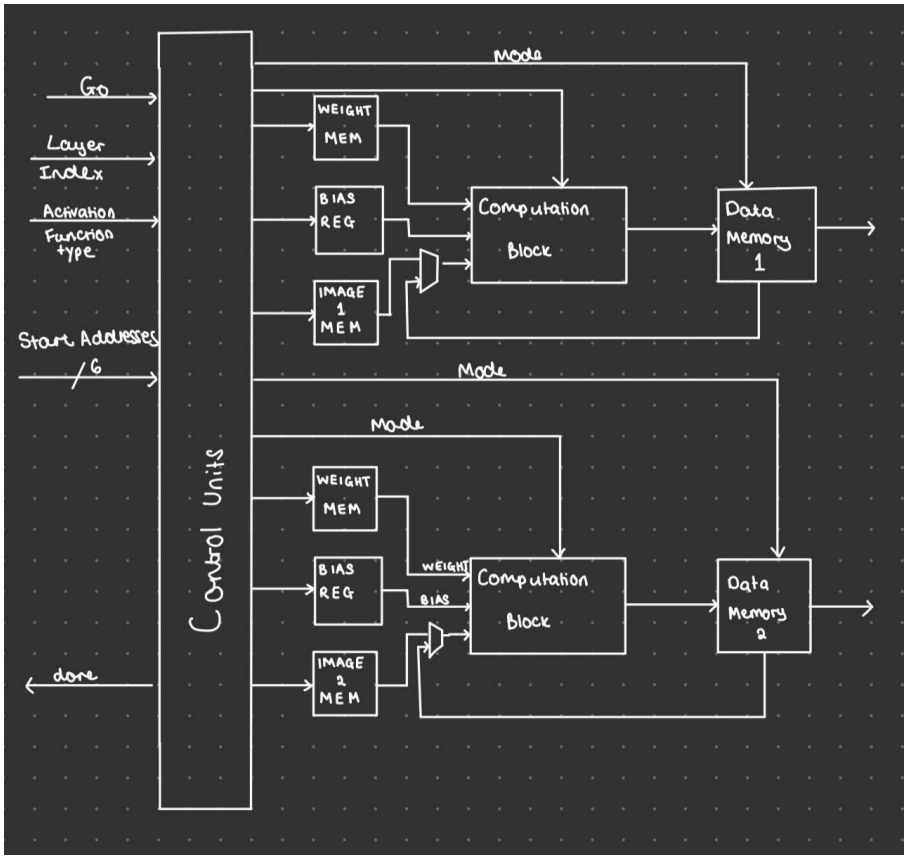
function towards the positive or negative side.[1] The main component of the hardware design would be the computation module. Our computation module would have the ability to perform a convolution layer, dense layer, max pooling layer, average pooling layer, spare layer and a series of activation functions. Initially, the focus would be on getting one set of these layers working but the final aim would be to have all of these working and interchangeable depending on the c file to configure the module.

The software side will have a C program that will configure the hardware side of the design depending on the neural network being implemented.

[1] ("Importance of Neural Network Bias and How to Add It")

Computation Module

The computational module is the main block on the FPGA side in our NN accelerator design. In order to enable high speed inference on the FPGA, we would design a module that is controlled by the on board software in real time. Initially, we will implement a computational module that performs all necessary computations for a CNN. However we hope to expand this design to include a multitude of different types of layer computations such as dense layer, sparse layer etc. This will enable us to take a more generic neural network design and accelerate it using the hardware. We also hope to take advantage of parallelism during this process so we will have multiple pipelines running simultaneously within this module. The microarchitecture for the computation module can be seen below:



We want to be able to run inference on multiple images at a time, hence why, as you can see above we hope to implement 2 pipelines. There will be a control unit (CU) that will dictate the state of the pipelines above. This will be the part that interfaces with the C program and as soon as the registers and memory blocks are loaded with the correct information will begin to drive the pipeline.

Initially, CU supplies the address from which the Image Memory (IMEM), Weight Memory (WMEM) and the Bias Registers (BREG) should be read from. The computation blocks take these inputs and perform a multiply and accumulate on the values. After 9 cycles (9 due to a 3 x 3 convolutional window), the computation module performs the required activation function on the value and writes it to the Data Memory (DMEM). This continues until we have performed the necessary convolutions on the entire image. This would be considered the first layer in a typical neural network. Instead of using the IMEM now we would utilize a mux and select the DMEM as the new input data source. Now we would perform the new calculations with the DMEM input data, new bias value and new weights from the WMEM. This would represent the second layer in our neural network. One key design feature of the computation block would be that it is both capable of performing convolution layer computations as well as pooling layer computations. This will be occurring within the same block.

Computation Block

The computation block is where we actually perform convolutions, pooling layer calculations and implement an activation function. The microarchitecture of the computation block can be described as a rotating series of line buffers that feed into a convolution module. Effectively, each line buffer stores the value of each row. As we are using a 3 x 3 kernel, we need 3 MEM arrays for the current calculations and 1 for the next row being loaded. On each clock cycle, we can push the pixel values of the first 3 rows into our convolution kernel. At the same time we will feed the values of the next row into the 4th MEM array. When the 4th MEM array is full and the 3 MEM arrays for the current calculation have been read all the way, the following happens:

- The 4th MEM array becomes the bottom row in the convolution kernel.
- The oldest MEM array so MEM array 1, now becomes the new loading MEM array as the information in this MEM array is no longer needed.

MEM array 1	CALCULATIONS
MEM array 2	
MEM array 3	
MEM array 4	LOADING ROW 4
NEXT STATE	
MEM array 2	CALCULATIONS
MEM array 3	
MEM array 4	
MEM array 1	LOADING ROW 5

Memory Resource Allocation

Memory Resource Requirements and Module Roles

The CNN accelerator design uses six total memory modules—two each for storing image data, convolutional weights (with bias parameters), and intermediate calculation results (data). Duplicating these modules allows us to enable dual channel processing // parallel computations throughout the pipeline.

Image Memory Modules (×1):

There is one 28 × 28 pixel MNIST image in every module. The MNIST images are typically in 8-bit grayscale, but in our design, we use them as binary values of either 0 or 1 but use 8 bits for each pixel to maintain normal resolution as defined by the data set. Such memories store the main input data for convolution operations as well as to perform secondary operations in processing.

- Per Module:
    - MNIST image size = 28 × 28 pixels = 784 pixels
    - Pixel bit length = 8 bits normally, 1 bit quantized in our design
    - Memory per module = 784 pixels × 1 bits = 784 bits (which equals 98 bytes)

Weight Memory Modules (×2):

These modules store weights needed for a given layer, along with associated biases. For different layers these modules must be different sizes, but they store a collection of int8 weights that correspond to the product of the input and output channels for Dense layers, or their kernel shape, input channels, and filters/output shape for convolutional layers. Each output channel also has a given bias. All weights and biases are stored in int8 data types.

For convolution model (Chollet):

Layer	Kernel Shape	in_channels	# Filters / out_channels	Weights	Weight Size	Bias Count (1/filter)
Conv2D #1	(3x3)	1	12	108	0.108KB	12B
Pool	(2x2)	12				
Dense	–	2028	10	20280	20.28KB	10B
Total					20.388KB	22B

To calculate number of weights = kernel shape * in_channels * filters

Data Memory Modules (×2):

Intermediate results and final outputs of CNN computations are stored in data memory modules. They offer direct access to computed values to be passed to subsequent stages of processing or to be stored for subsequent post-processing. With their dual banks of data memory, the design supports concurrent streams of processing, increasing overall throughput and efficiency of the accelerator. For memory constraints, it is important to consider these too on a per-layer basis, mainly focused on the product of the output sizes with the int8 data type for the weights.

Total Memory Considerations/Requirements:

- Image Memory = 196 bytes

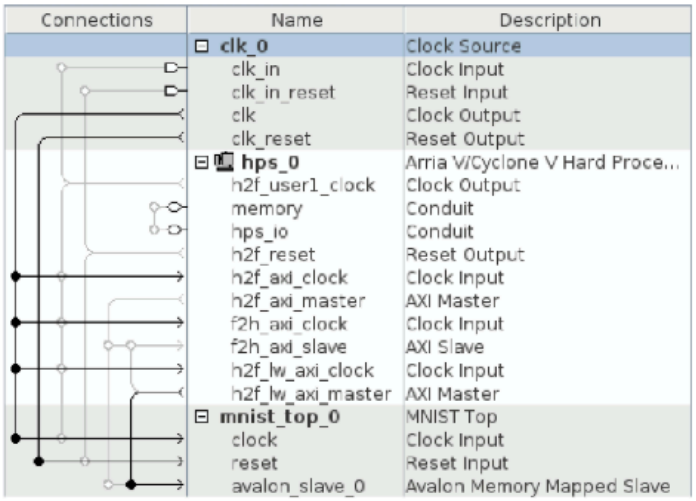
- Weight + Bias Memory = 16.288KB
- Data Memory (assuming pipelined worst case) = 2KB
- Total Memory Req = 18.4KB

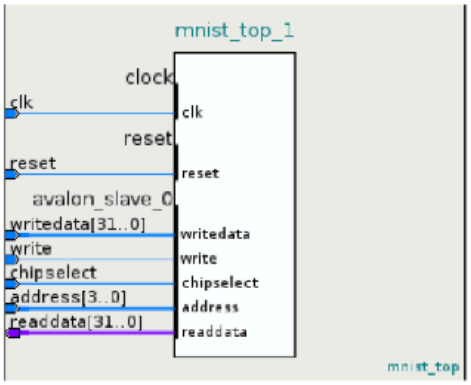
Hence, we can see that the total system memory requirements for the computationally challenging convolutional network are acceptable at **18.4KB**, leaving a significant amount of memory for other processing tasks within the system design. For the simple to compute dense layer ‘simple’ model, we can batch weights in sequentially for a dense layer much more easily than a convolutional layer, allowing us to not store all weights in memory at the same time.

Hardware/Software Interfaces

3 Pillars of Communication

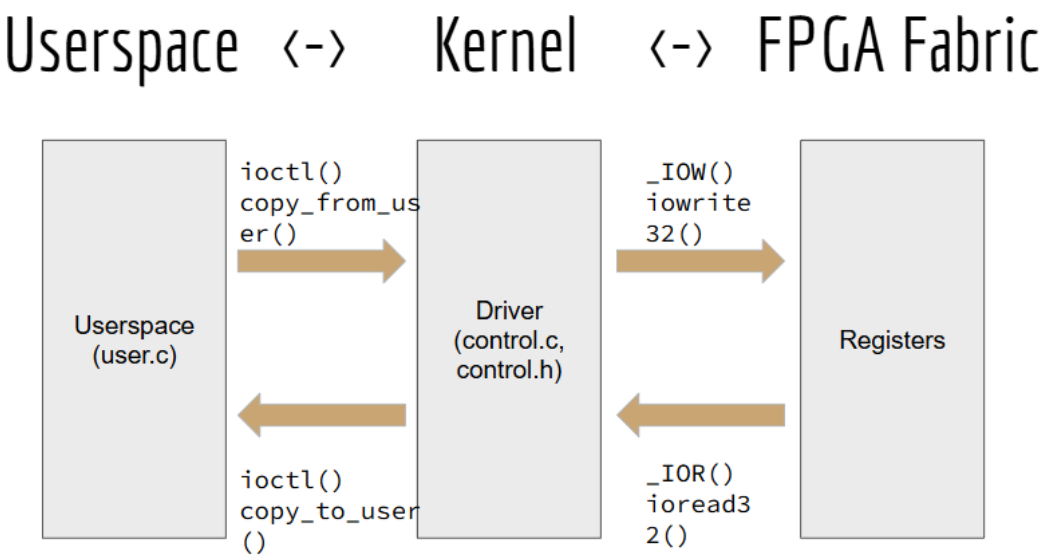
One can think of the interface as communication across 3 different realms: the userspace (user.c), kernel space (control.c, control.h), and hardware registers. The point of contact with the hardware is the top-level verilog module (top.sv) with signals corresponding to those of an Avalon memory slave (making initialization and wiring in Platform Designer much easier). This module then initializes instances of the other hardware components.





The registers are accessed by the driver module through `iowrite32()` and `ioread32()` calls (which take in the register addresses as a parameter), and each call has their own opcode corresponding to its purpose; this will be further discussed in the "Communication Protocol" section. The userspace then utilizes said functions defined in the driver module to make these iowrite/read calls using `ioctl()` function calls, which take in the characteristic opcode as a parameter. For data transfer between the userspace and driver files, the copy_from_user() and copy_to_user() functions–which take in data buffer addresses for the information being

sent/received from hardware as parameters–are called within the `ioctl()` function handler of the driver.



Register Map

Below is the register map that the software interprets as registers in what is essentially the FPGA fabric. All registers are 32 bits wide, even if not all the bits hold valuable information (we consistently call `iowrite32()` or `ioread32()`). The first register, at the Avalon register address with offset 0, is used to carry the constructed word images. The following 10 registers, at offsets 1-10, are used to carry the final dense classification outputs, ie. the probability of each digit in sequential order (0-9). Finally, the last register is used by the hardware to signal the userspace through the driver that it is ready to begin sending the aforementioned dense output signals through a 'go' bit.

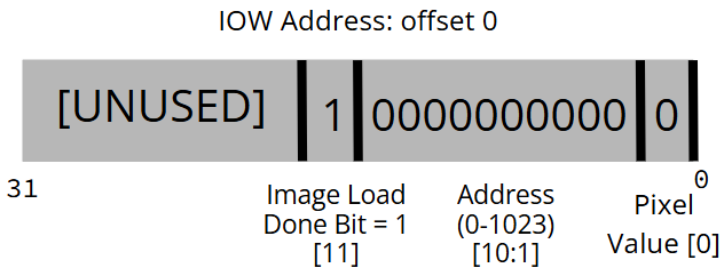
Register # (Offset / Addr)	Function	Code Definitions
0	Constructed Image Word	`#define IMAGE(x) (x)`
1	Dense Classification 0	`#define PROB_0(x) ((x)+4)`

2	Dense Classification 1	`#define PROB_1(x) ((x)+8)`
3	Dense Classification 2	`#define PROB_2(x) ((x)+12)`
4	Dense Classification 3	`#define PROB_3(x) ((x)+16)`
5	Dense Classification 4	`#define PROB_4(x) ((x)+20)`
…….	*Dense Classifications 5-7*	
10	Dense Classification 8	`#define PROB_8(x) ((x)+36)`
11	Dense Classification 9	`#define PROB_9(x) ((x)+40)`
……	[EMPTY]	
15	Dense Done Bit	`DENSE_DONE_F(x) ((x)+60)`

Communication Protocol

Throughout the processing of one image by the CNN, 4 main occurrences of data transfer happen, each represented and executed by an `ioctl()` call:

1. `ioctl(control_fd, CONTROL_WRITE_IMAGE, &vla)`: Software is first used to load in the image (stored as a .mem file) into a buffer, append other necessary information to the bit to form a 12 bit word (pixel bit address, "last image pixel" flag), and then send the word to the hardware side which will then send it to the convolver layer.
    a. This is an `iowrite32()` call under the hood at offset 0:

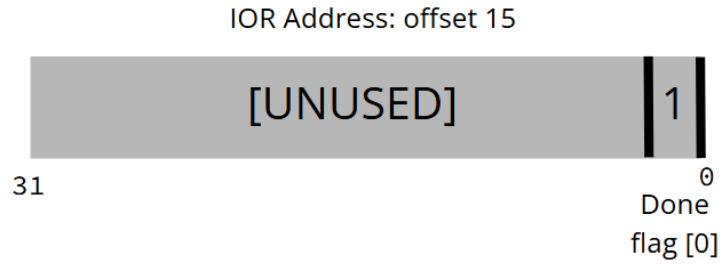


b. Each pixel of the image is 1 bit, being either black or white

c. On the last pixel, the "image load done bit" is set high to flag the hardware to start running the convolution layer

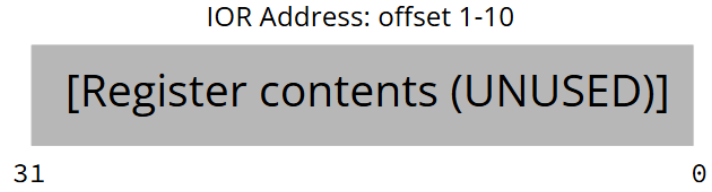
2. `ioctl(control_fd, CONTROL_READ_DENSE_DONE, &vla)`: After sending the image data, the software continuously polls for the flag indicating that the hardware is done with its calculations (ie. when the dense layer is finished)

   a. This is an `ioread32()` call done on the register at offset 15:

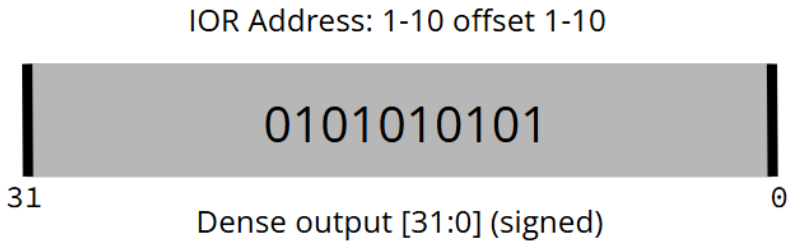


3. `ioctl(control_fd, CONTROL_TRIGGER_DENSE_ADDR, &vla)`: Once the flag is seen, software then sends an `ioread32()` call on each of the "Dense Classification" registers (offests 1-10) in order to let hardware know where to send the dense layer outputs

   a. This is an `ioread32()` call done on the registers at offsets 1-10, but since we are simply using this call as a "trigger" (to let hardware know where to place the data), we don't care about the register contents and hence the readdata is not stored in a buffer:



4. `ioctl(control_fd, CONTROL_TRIGGER_DENSE_ADDR, &vla)`: Finally, after hardware inputs the 10 dense results in their corresponding registers (done 1 clock cycle after receiving the previous `ioread32()` call), software does another `ioread32()` sweep across registers at offsets 1-10 to grab the CNN results and print them out to the user
    a. This is another `ioread32()` call done on the registers at offsets 1-10, but this time the values are stored in a buffer to be sent back up to userspace



Example function path from image data in userspace to hardware register:

user.c:

1. Call `load_image_data()` in `main()`

```c
/* run fgets(), build image_word and send with ioctl */
while (fgets(line_buff, sizeof(line_buff), im_fp) && pixel_count < IMAGE_SIZE) {
  /* Create image word to be sent */
  uint32_t image_word = 0;

  /* Get building blocks of image word */
  uint8_t flag = (pixel_count == IMAGE_SIZE - 1) ? 1 : 0;
  uint16_t addr = pixel_count;
  uint8_t pixel_data = (line_buff[0] == '1') ? 1 : 0;

  /* Build image word */
  image_word |= (flag & 0x1) << 11; // make sure flag is 1 bit, shift it left 11 bits, and then 'or' it with image_word
  image_word |= (addr & 0x3ff) << 1;
  image_word |= (pixel_data & 0x1);

  /* Send image with ioctl */
  if (load_image_data(&image_word) != 0) {
    perror("ioctl(CONTROL_WRITE_IMAGE) failed");
    return EXIT_FAILURE;
  }

  pixel_count++;
}
```

2. `load_image_data()` definition in user.c

```c
int load_image_data(uint32_t *image_data_word) {
  control_arg_t vla;
  vla.image_data = *image_data_word;
  if (ioctl(control_fd, CONTROL_WRITE_IMAGE, &vla)) {
    perror("ioctl(CONTROL_WRITE_IMAGE) failed");
    return -1;
  }
  return 0;
}
```

3. `ioctl()` handler in control.c

```c
static long control_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    control_arg_t vla;      You, 3 days ago • Driver finished, need to code us

    switch (cmd) {
    case CONTROL_WRITE_IMAGE:
        if (copy_from_user(&vla, (control_arg_t *) arg,
                sizeof(control_arg_t)))
            return -EACCES;
        write_image_data(&vla.image_data);
        break;
```

4. iowrite32() to respective register in control.c

```c
static void write_image_data(uint32_t *image_data)
{
    iowrite32(*image_data, IMAGE(dev.virtbase));
    dev.image_data = *image_data;
}
```

Testing the Accelerator

There are two main levels of verification for this accelerator:

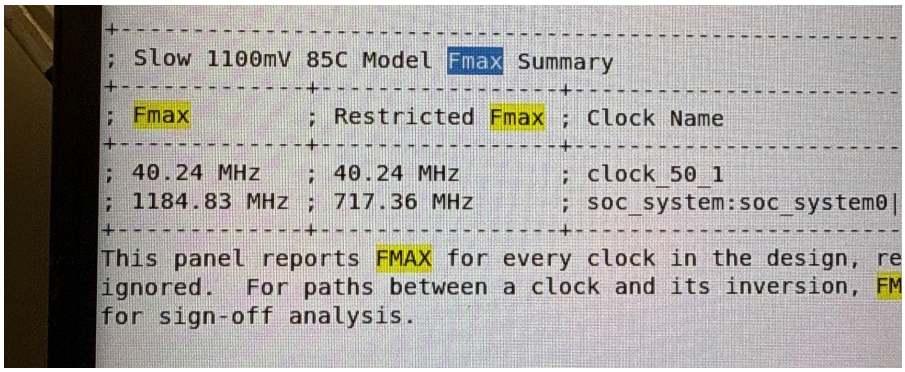
1. Functionality: comparing the accuracy of the accelerated system when performed on a baseline laptop system.
2. Performance: comparing the speedup of the system compared to performing all operations in software on the same system.

The functionality benchmark is essential. By finding which value of the final Dense layer is the maximum value, we can determine which of 10 digits the model predicts a given image to represent. This can be checked against the label of that image: if the prediction matches the label, then the model is correct. Otherwise, the model is incorrect. Accuracy is defined as the number of correct tests divided by the total number of tests. For this system, the accuracy should not just be strong, but should match a laptop system's accuracy for the same model: this will ensure the accelerator system is performing the proper tasks.

The performance benchmark should be measured at minimum on the De1-SoC in a vacuum, evaluating how quickly the model runs on this system. Ideally, this verification would include a direct comparison to a model carrying out all computations on the De1-SoC in C software, and compare the speedup.

In regards, to functionality and performance, we were able to successfully demonstrate the following:

- Bit-packed and processed image data loading.

- Successful translation of data across hardware and software through kernel/header software.
- Status polling for hardware completion flags and FSM for state.
- Retrieval of final 32-bit classification probabilities for each possible digit.
- Total Operational Time: Ranges from 1.01-1.07 milliseconds @ roughly 30% resource utilization.
- 


```
root@de1-soc:~/mnist-accelerator/sw# ./user
Control Userspace program started
Dense output 0: -67
Dense output 1: -77
Dense output 2: -102
Dense output 3: -79
Dense output 4: -143
Dense output 5: 24
Dense output 6: -143
Dense output 7: -114
Dense output 8: -205
Dense output 9: 0
Control Userspace program terminating
```

Bibliography

"Importance of Neural Network Bias and How to Add It." *Turing*, https://www.turing.com/kb/necessity-of-bias-in-neural-networks#what-is-bias-in-a-neural-network? Accessed 15 April 2025.

Chollet, François. Keras Documentation: Simple MNIST Convnet. 19 June 2015, https://keras.io/examples/vision/mnist_convnet/.

Kizheppatt, Vipin. *Neural Networks on FPGA: Part 2: Designing a Neuron*. 2021. *Youtube*, https://www.youtube.com/watch?v=a2wOjxRf_xg&t=488s. Accessed 15 April 2025.

"MNIST database." *Wikipedia*, https://en.wikipedia.org/wiki/MNIST_database. Accessed 15 April 2025.

Patel, Prathmesh, et al. *Acceleration of Digit Classification Using Custom CNN on a SoC FPGA*. 2024.

"Training a Neural Network on MNIST with Keras | TensorFlow Datasets." *TensorFlow*, 14 December 2024, https://www.tensorflow.org/datasets/keras_example.

Quantization. https://huggingface.co/docs/optimum/en/concept_guides/quantization. Accessed 15 May 2025.