ECE 498NSU Fall 2020

Final Project: SqueezeNet Fourth Firemodule

Alkin Ozyapici
Abhi Kamboj

Table of Contents:

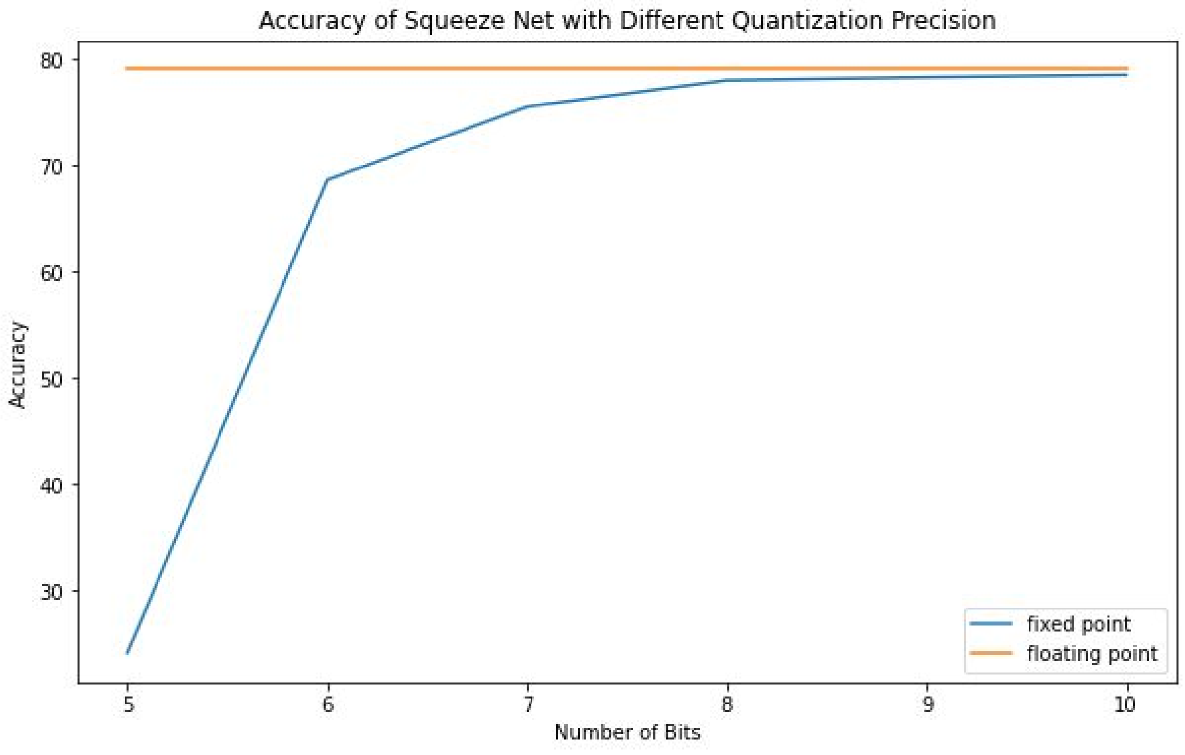
Step 1 (Fixed-point analysis)

The pre-trained weights and biases were used to determine the quantization bit-precision and the clipping value for the inputs, outputs, weights and biases.

For the weights and biases the clipping range is (-1,1) by default as these tensors are always in this range. For the inputs and the outputs the clipping range was determined by trial and error. As the outputs are accumulation of the product of several weights and inputs, the clipping range for the outputs were set to be 4 times the clipping for the inputs. After the trials the optimum clipping value for the inputs was determined to be 2, while the optimum clipping value for the outputs was determined to be 8.

For hardware implementation ease, uniform quantization was determined to be better for the entire net. The number of bits was swept from 5 to 10, and the inference accuracy of the network can be seen in Figure 1. From this data the optimum quantization precision was determined to be 8 bits. While 8 bits do not result in an accuracy strictly 1% off of the floating point precision, the improvements after 8 bits were insignificant and the trade off adding additional bits was determined to be unnecessary because of the increased hardware cost.

Figure 1:



Step 2 (Architecture Design)

PE Design

Each processing element (PE) was designed with a multiplier, adder and register. The multiplier and adder form the multiply accumulate (MAC) unit and the register stores the accumulated result (Figure 2). The PEs were aligned in an array of 9 elements to perform an output stationary computation (Figure 3). In addition, there is an accumulator that adds all the PEs for the 3x3 convolution. All the PEs have access to memory and inputs and weights are loaded in parallel as shown in the diagrams below.

Figure 2: Internal View of a PE Unit

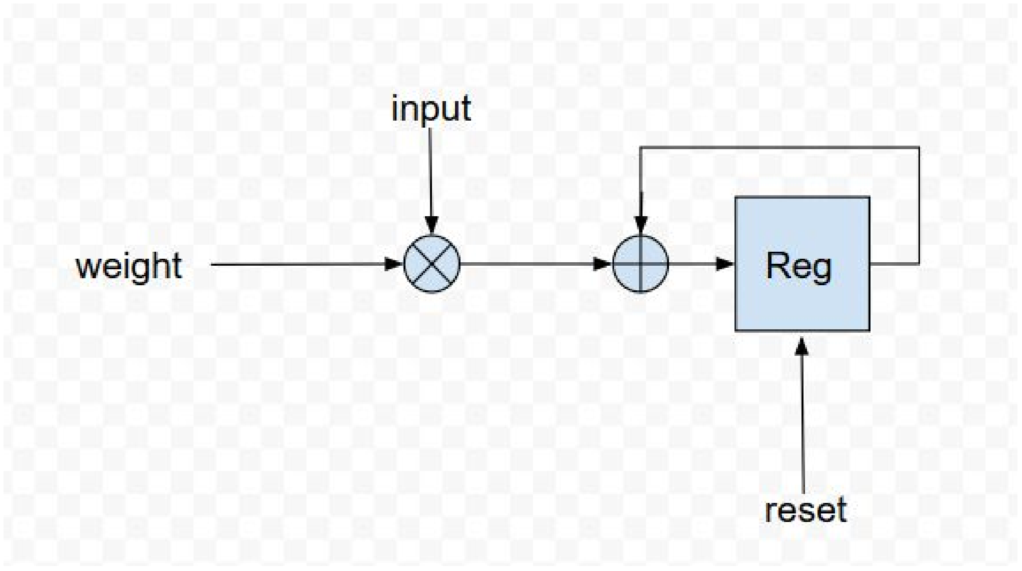
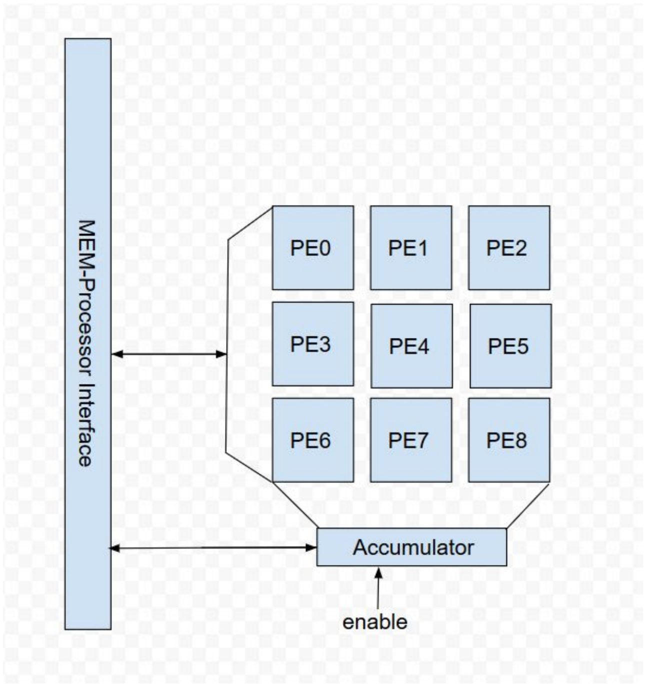


Figure 3: The PE Unit Architecture Diagram



We decided to use output stationary as opposed to weight stationary because the number of pixels in an output is consistent throughout the fire module. As shown in Figure 4, in the convolution visualizations, all the outputs are Nx3x3 blocks, where N is the number of output channels. If the PE array were designed to be weight stationary, for the Squeeze 1x1, we would need 256 PEs, but for the Expand we would need 32 for the 1x1 and 32x9=288 for the 3x3. If we use 200+ PEs, it would be a very large processing unit and would consume lots of energy. In addition, since the Expand 1x1 layer only needs 32 weights for a convolution, about 170+ PEs would be wasted in the weight stationary. These could be routed to process multiple kernels at once, but this would make the system unnecessarily complex. Although we might get a high decision throughput using more PEs, the energy costs would be very large and wasteful since there would be a poor resource utilization and there would be a much higher chance of bugs because of complexity.

Alternatively, we could have implemented weight stationary processing by holding just the filter kernel at one input channel (so the PE array would be the size of the 2D filter kernel) but then we would only need 1 PE for the 1x1 convolutions. This defeats the purpose of having a systolic architecture as only one element is being processed at a time. This would result in a very high decision latency and low throughput, as it would take many more clock cycles since only one element of the convolution is being processed at once.

Another design we considered was input stationary where the MACs and kernels rotated rotated through the PEs and the inputs stayed the same. The issue with this is that then each PE would have to store multiple MACs, one for each filter kernel. This results in utilizing much more storage and hardware than necessary and may increase latency as it requires more reads and writes.

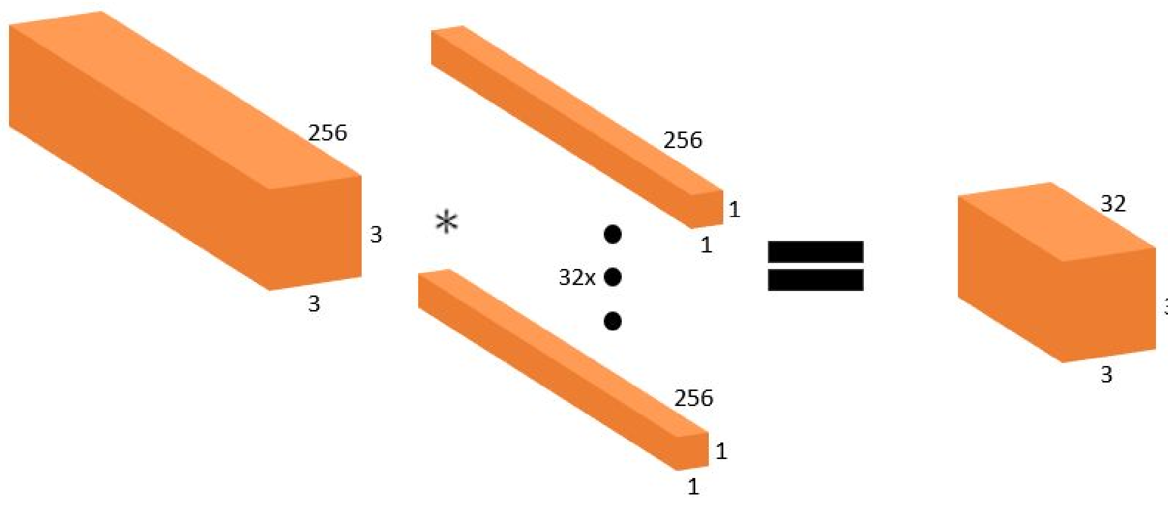
In our output stationary design, for the 1x1 convolutions we hold the MACs stationary at each PE and rotate the weights for a single kernel and the inputs into the PEs. After 256 clock cycles for squeeze1x1 (or 32 clock cycles for expand1x1), we add the bias and we will have the output for one output channel, then we can repeat for all the filters. For the 3x3 convolution, we used the fact that the convolution filter conveniently matches the size of the output to implement a output-pixel stationary method that uses all 9 PEs to calculate the output for one pixel as opposed to 9 pixels at once like the 1x1 convolutions. Our methods are further described in the "State Machine and PE Use" section.

Our output staionary method does not require more than one MAC to be stored in the PE and all of the PEs are being used during every clock cycle. Hence, this provides for the best balance between energy consumption and latency or decision throughput.

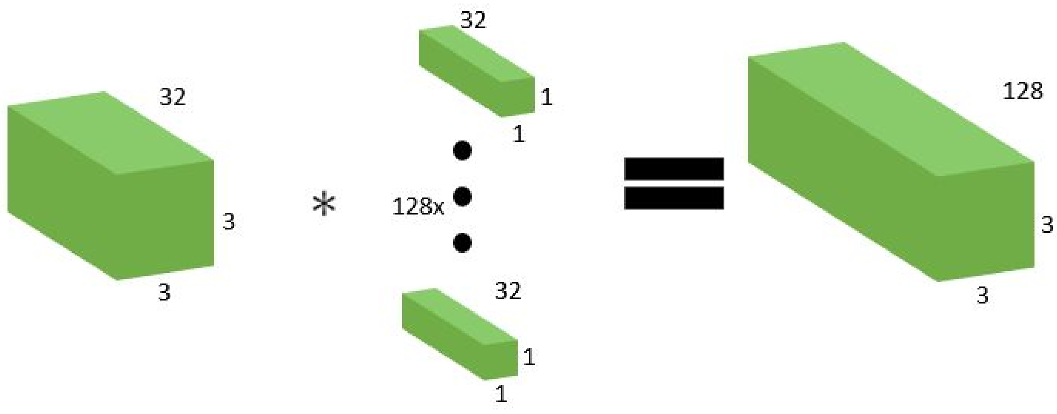
Figure 4: Visualizations of Convolutions implemented in hardware:

- Note: biases are not shown but are added to each convolution, so they do not change the dimensionality of the images

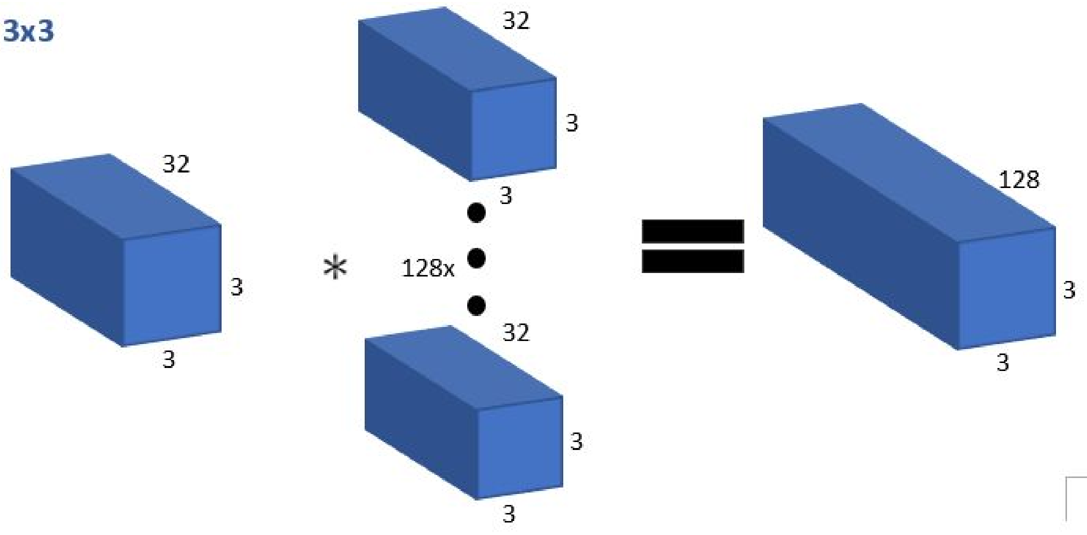
Squeeze 1x1



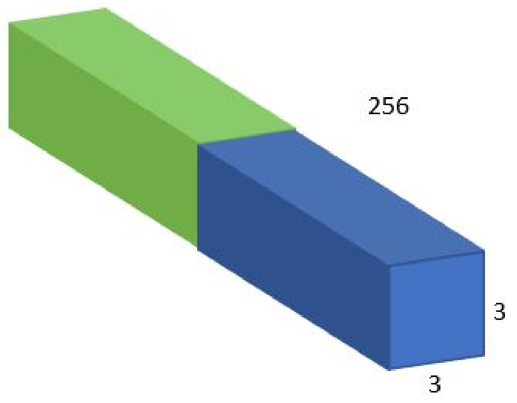
Expand 1x1



Expand 3x3



Output: (Concatenated Expand outputs)



State Machine and PE Use

The fire module has 3 steps: the 1x1 squeeze convolution, the 1x1 expand convolution, and the 3x3 expand convolution. In the 1x1 convolutions, each PE represents 1 output pixel, and for the 3x3 convolution each PE represents a given filter kernel's partial accumulation. Below are explanations of the convolution followed by the state diagram (Figure 5).

The 1x1 convolution works as follows:

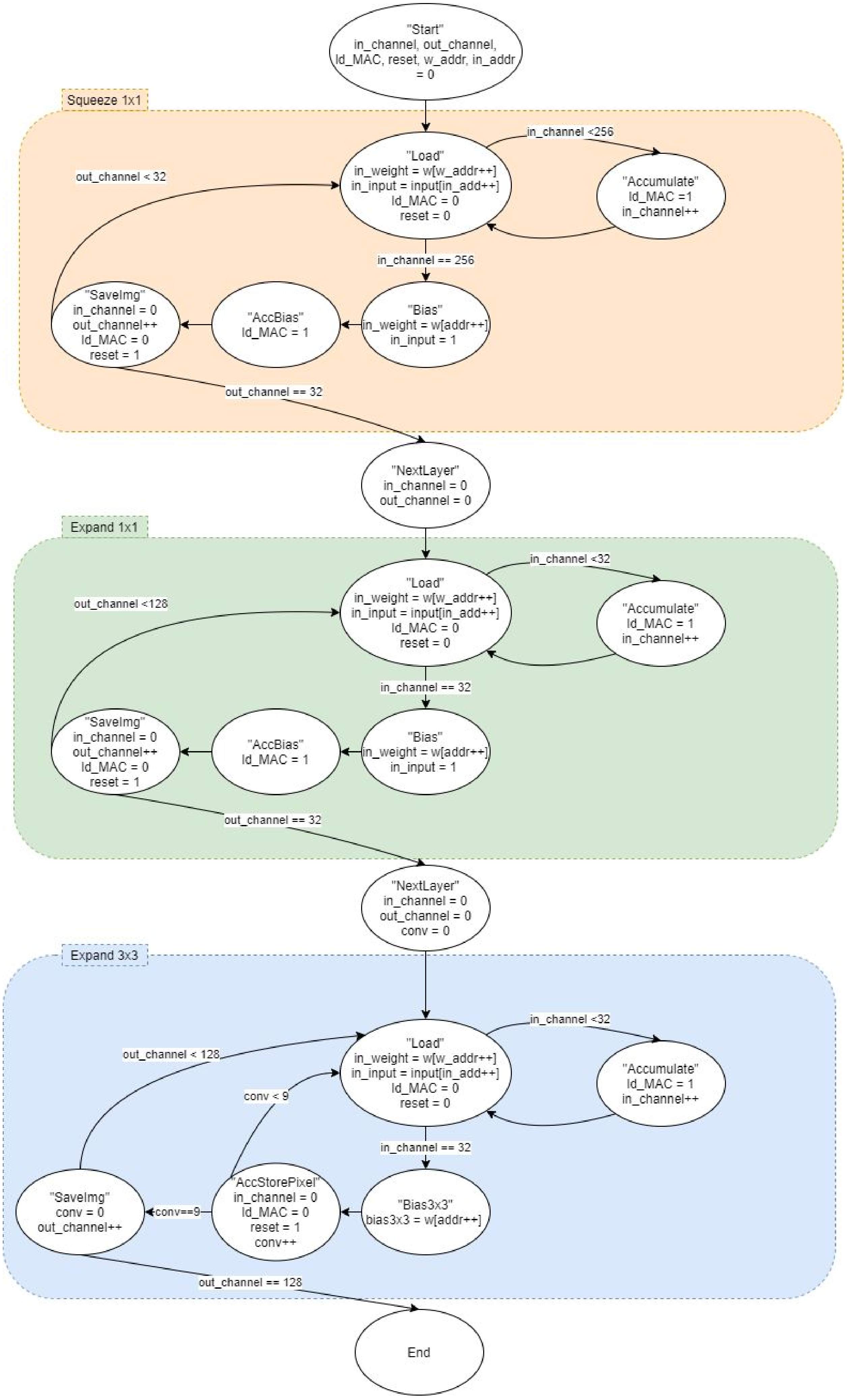
1. Start:
    a. initialize 2 counters: in_channel and out_channel to zero
    b. Initialize addresses, load_MAC, and reset to zero
    c. Note: in the state diagram "w" is the weight RAM where the weight values are held and "input" is the input RAM where the inputs are stored.
2. Load:
    a. Broadcast the in_channel'th weight of the out_channel'th filter across all PEs
        i. Note: the same weight goes to every PE for the 1x1 convolution.
    b. Load the first inputs to each PE, in parallel.
        i. Note: Each PE has a different input.
3. Accumulate:
    a. Each PE multiplies the weight and input and accumulates it in a local register when ld_MAC is high.
    b. Increment the in_channel counter
    c. Repeat steps 2-3 for all input channels, (until in_channel counter reaches the number of channels). At every Load, the next set of weights and inputs are being fed from the memory to the PE. The memory is assumed to be loaded with the correct values in the correct order such that it can be read sequentially. This means the inputs are in row major order from the first input channel to the last, and the weights of each filter are listed with the biases in between the last weight of one filter and the first weight of the next filter.
4. Bias:

    a. Broadcast the bias to each pixel through the weight
    b. Broadcast 1's as the input
5. AccBias:
    a. Add the bias to each output pixel
6. SaveImg:
    a. Set reset = 1, to clear the PEs.
    b. At this point one Nx1x1 filter has been convolved with Nx3x3 block forming one 3x3 output, where N is the number of input channels. Save this 3x3 image as the first output channel and repeat steps 2- 6 for all filter kernels.

The 3x3 convolution functions as follows:

1. Next Layer:
    a. Initialize in_channel, out_channel, and conv to zero
2. Load:
    a. Load each weight for the first 3x3 filter. Each PE will hold a different weight
    b. Load each input. Each PE will hold a different input.
3. Accumulate:
    a. Each multiply and accumulates each in a local register when ld_MAC is high
    b. Repeat steps 2-3 for all input channels, so until the in_channel counter reaches the number of channels. This step is the same as for the 1x1 convolution, except all the PE's are accumulating for pixel output. Also note, that the memory stores the inputs with padding sequentially as used by this control unit.
4. Bias3x3:
    a. Instead of using the PEs to add the bias, the fire module takes in a separate bias signal called bias3x3 (bias3x3 is not shown in Figure 3, for clarity but can be seen in the code in Appendix B). Alternatively, the bias can be read as another weight on a filter with inputs = 0 everywhere besides the center pixel (so it would appear in all convolutions).
5. AccStorePixel
    a. The bias3x3 signal is added with the value being stored in every PE to produce one output pixel and this output pixel value is stored.
    b. Set reset = 1 to clear all the PEs MAC registers.
    c. Repeat steps 2-5 for the remaining 8 dot-products for this filter.
6. SaveImg:
    a. At this point one Nx3x3 filter has been convolved with one Nx3x3 block forming one 3x3 output, where N is the number of input channels. Save this 3x3 image as the first output channel and repeat steps 2- 6 for all filter kernels.

Figure 5: State Machine


"Start" in_channel, out_channel, ld_MAC, reset, w_addr, in_addr = 0
Squeeze 1x1
"Load" in_weight = w[w_addr++] in_input = input[in_add++] ld_MAC = 0 reset = 0
in_channel <256
"Accumulate" ld_MAC =1 in_channel++
out_channel < 32
in_channel == 256
"SaveImg" in_channel = 0 out_channel++ ld_MAC = 0 reset = 1
"AccBias" ld_MAC = 1
"Bias" in_weight = w[addr++] in_input = 1
out_channel == 32
"NextLayer" in_channel = 0 out_channel = 0
Expand 1x1
"Load" in_weight = w[w_addr++] in_input = input[in_add++] ld_MAC = 0 reset = 0
in_channel <32
"Accumulate" ld_MAC = 1 in_channel++
out_channel <128
in_channel == 32
"SaveImg" in_channel = 0 out_channel++ ld_MAC = 0 reset = 1
"AccBias" ld_MAC = 1
"Bias" in_weight = w[addr++] in_input = 1
out_channel == 32
"NextLayer" in_channel = 0 out_channel = 0 conv = 0
Expand 3x3
"Load" in_weight = w[w_addr++] in_input = input[in_add++] ld_MAC = 0 reset = 0
in_channel <32
"Accumulate" ld_MAC = 1 in_channel++
out_channel < 128
conv < 9
in_channel == 32
"SaveImg" conv = 0 out_channel++
conv==9
"AccStorePixel" in_channel = 0 ld_MAC = 0 reset = 1 conv++
"Bias3x3" bias3x3 = w[addr++]
out_channel == 128
End

Step 3 (Mapping to Hardware)

Hardware Implementation

After the architectural design the PE array was implemented in System verilog. The hardware implementation of the singular PEs and the PE array with ReLU implementation can be found in Appendix B. For the hardware implementation the clipping value for the inputs was increased to 8 for simplicity, since outputs of the first convolution are later fed into the other two layers, and the output clipping value was determined to be 8 in Step 1. Setting the input and output clipping value to be the same simplified the hardware implementation and testing.

One problem that arose during the hardware implementation that is abstracted in the software is bit growth during multiplication and accumulation. Every time two fixed point numbers are multiplied and added the precision of the result increases. To account for this increase we used the Bit Growth Criterion as derived in class: $B_y = B_x + B_w + \log_2(N)$, where Bx = 8, Bw = 8 and N = 256 (for the squeeze1x1 256 channel dot product). Hence By = 24, so the MAC register in the PE unit held a 24-bit fixed point value. As the output from the fire module needed to be 8 bits, the PE 24-bit output was quantized back down to 8 bits before being returned from the module. This assured that the module still performs the computations within the correct accuracy while maintaining the 8 bit quantization.

Due to the lack of a physical FPGA board, the M/P interface for this project was implemented in the testbench as a large array that fed in values to the fire module. However, this can be easily realized on an FPGA board by initializing the RAM memory. Ideally, inputs and weights would have been read from RAM and the outputs would have also been stored in the RAM. Through Vivado or Quartus an IP block of RAM memory could have been instantiated. The memory would have been initialized to the correct values from a memory file or text file generated from the python, using the Vivado/Quartus GUI or the $readmemh command in verilog. Then as described in the State Machine above, the logic unit could simply control the weight and input RAM addresses being fed into the fire module. The amount of memory needed for can be seen in Figure 4. For the input RAM 256x3x3 = 2304 bytes is needed. The memory needed for the weight RAM is 256x32x1x1 + 32x128x1x1 + 32x128x3x3 = 49152 bytes. For the bias RAM: 32 + 128 + 128 = 288 bytes. Finally, the outputs would need 256x3x3 = 2304 bytes, but these outputs could be written over the input RAM to conserve space. Overall we would need about 2304 + 49152 + 288 = 51744 bytes of RAM (excluding padding) which is about 51KB. According to this website: https://www.tul.com.tw/ProductsPYNQ-Z2.html the Pynq-Z2 board we would have used for this project has 630 KB of RAM, which should be more than enough for this module design. In addition, since it has a 16-bit bus, and we only need to feed 16 bits at a time, 8 for the inputs and 8 for the weights, we could theoretically run this design at a maximum speed of 1050Mbps, as that is the access speed of the board and memory access is the most critical delay in latency.

Hardware Testing

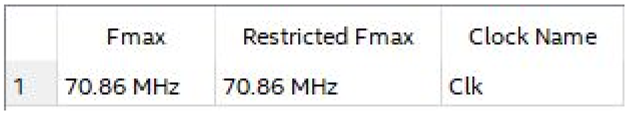
Before the inputs from the software models were fed into the array, the PE unit was tested with a length 10 dot product to assure the functionality of the PE arrays.10 length input and weight vectors were initialized and fed into all 9 PE units. The PE array passed the test for the test vectors, verifying the functionality of the design. The test code used for this step and the waveform output can be found in Appendix C. Also shown in Appendix C is the modelsim waveform. As seen in this timing diagram, the ideal_output value matches the sum of the PE_outputs, indicating that the PE array is working as expected.

After the design was verified, the above described state machine was implemented in a testbench, and the extracted inputs, weights and biases were fed into the PE array accordingly. After continuous testing the output of the first convolution was acquired correctly from the hardware. This further proves that the hardware is capable of performing 257 length dot products, including the bias. However, we were unable to get the final layer output correctly, presumably due to implementation errors of the state machine in the testbench. This was due to the limited time and the wrong implementation, rather than the capabilities of the PE array implementation since the PE array is able to perform a 257 length product without errors.

Step 4 (Mapping Evaluation)

Synthesis Metrics

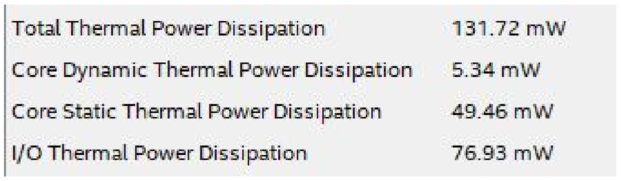
Clock Frequency Data



	Fmax	Restricted Fmax	Clock Name
1	70.86 MHz	70.86 MHz	Clk

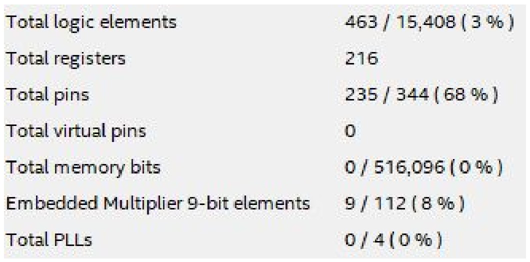
From the simulations of Quartus Prime, the maximum frequency of the design was found to be 70.86MHz. This is reasonable since the design implements a one cycle 8-bit by 8-bit multiplier and it is expected that this will limit the clock frequency of the design.

Power Data



Total Thermal Power Dissipation	131.72 mW
Core Dynamic Thermal Power Dissipation	5.34 mW
Core Static Thermal Power Dissipation	49.46 mW
I/O Thermal Power Dissipation	76.93 mW

Resource Utilization



Total logic elements	463 / 15,408 (3 %)
Total registers	216
Total pins	235 / 344 (68 %)
Total virtual pins	0
Total memory bits	0 / 516,096 (0 %)
Embedded Multiplier 9-bit elements	9 / 112 (8 %)
Total PLLs	0 / 4 (0 %)

Results and Conclusion

Throughput and Latency for the 4th Fire Module - Conclusion

From the data it can be seen that the design can run at 70MHz and every PE can complete 1 MAC per cycle. The first squeeze convolution will be completed in 8256 when the state transitions in the above diagram is implemented correctly. The expand1x1 convolution will be completed in 4352 cycles, and the expand3x3 convolution will be completed in 39168 cycles. In total to generate the output for one input, the design will take 51776 cycles which results in a layer latency of 0.73ms and a throughput of 1351.98 outputs per second. Using the power data and the layer latency, the energy per decision for this layer is 96.1556uJ.

Overall, implementing this originally floating point layer in fixed point will result in significant energy savings, since the number of MACs is significantly reduced compared to the general 32bit floating point numbers. Furthermore, there are a total of 8 Fire modules in SqueezeNet. Implementing all these Fire modules in 8-bit fixed point hardware will yield significant per decision energy savings. Also implementing all these layers, in fixed-point hardware will decrease the storage recruitment for the networks as the bit-size for the variables are four times smaller than the floating point implementation.

Discussion on limitations and improvements:

For hardware simplicity the clipping value was kept the same for inputs and outputs, unlike our software analysis with clipping value of 2 for inputs and 8 for outputs. These clipping ranges can be implemented in hardware to yield the optimum accuracy for the network that was acquired in software modeling.

In our design of the PEs, our goal was to balance the energy consumption and throughput, however, depending on the limitations of the system alternative designs may have been better. For example, if the design needed to be really fast, but energy consumption or hardware complexity was not an issue, our PE design could have been extended to a 3 dimensional PE array of 3x3x32. In this case, the fire module could have computed all the 32 filters' dot products in just 256 clock cycles for the Squeeze1x1. For Expand1x1 layer it would take 128 clock cycles, and for Expand3x3 128x9=1152 clock cycles. 256+128+1152= 1536 cycles is much faster than 51776 which is what was calculated for our design, however since this design is 32 times larger than ours it may take an upper bound of 32*96 = 3072uJ, or 3mJ of energy. Therefore, this design is much faster than our original design however would require a much larger energy per decision.

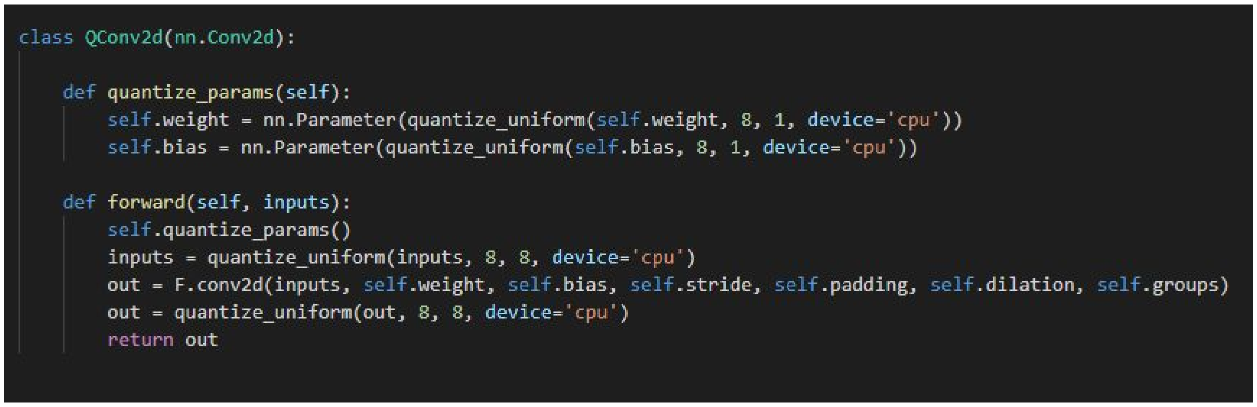
Contributions of Each Team Member:

Alkin Ozyapici: Worked on software modeling the SqueezeNet and acquired the quantization precision and clipping values for the network. Extracted the parameters of the 4th Fire module from the software prototype. Verified the hardware and implemented the state machines in the testbench.

Abhi Kamboj: Worked on creating the PE design and realizing it in hardware. Explained why this design is better than other potential designs in the report, and created the convolution visualization and State Machine diagram.

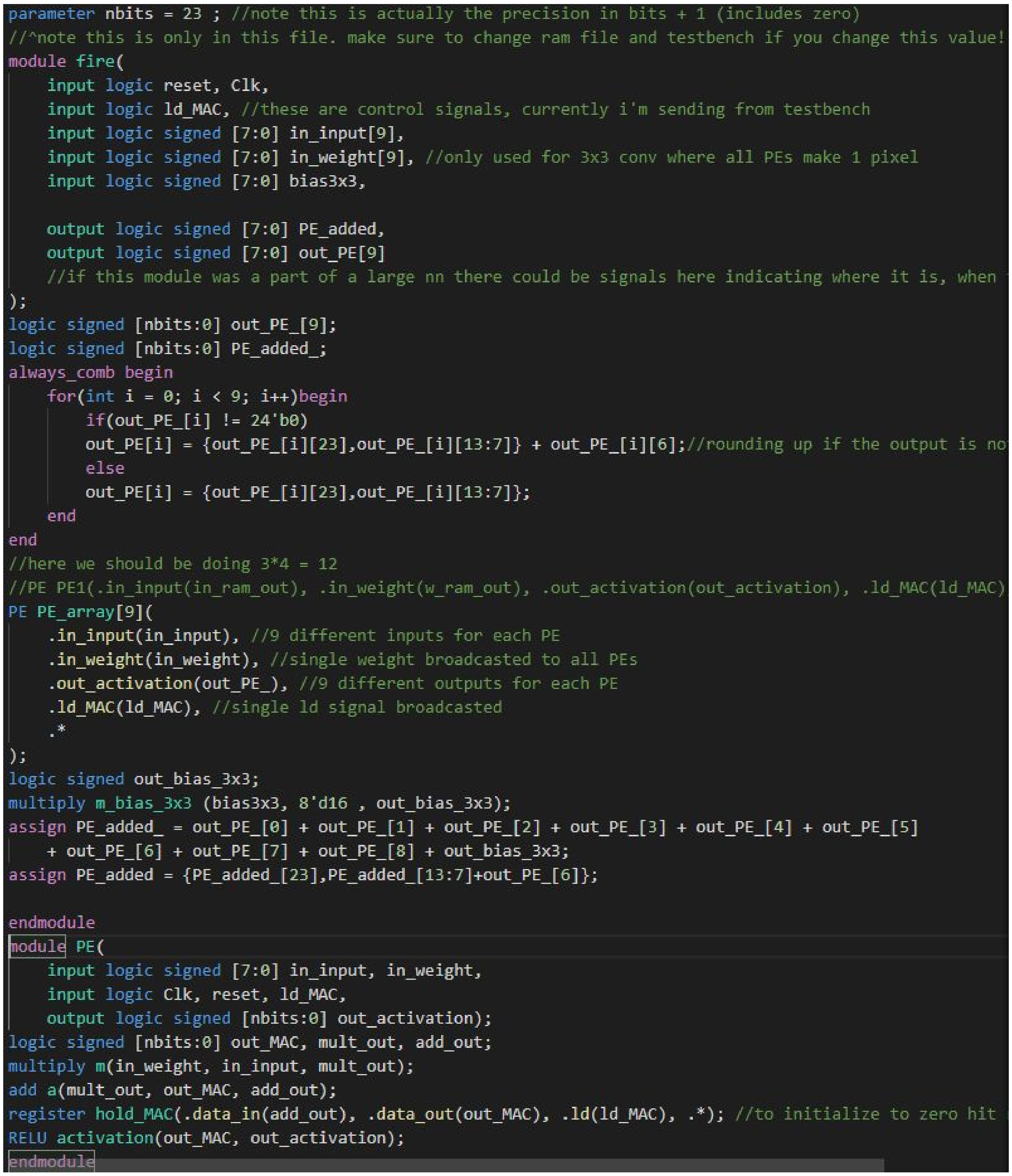
Appendices:

Appendix A: Fixed Point Conv2d Layer Code

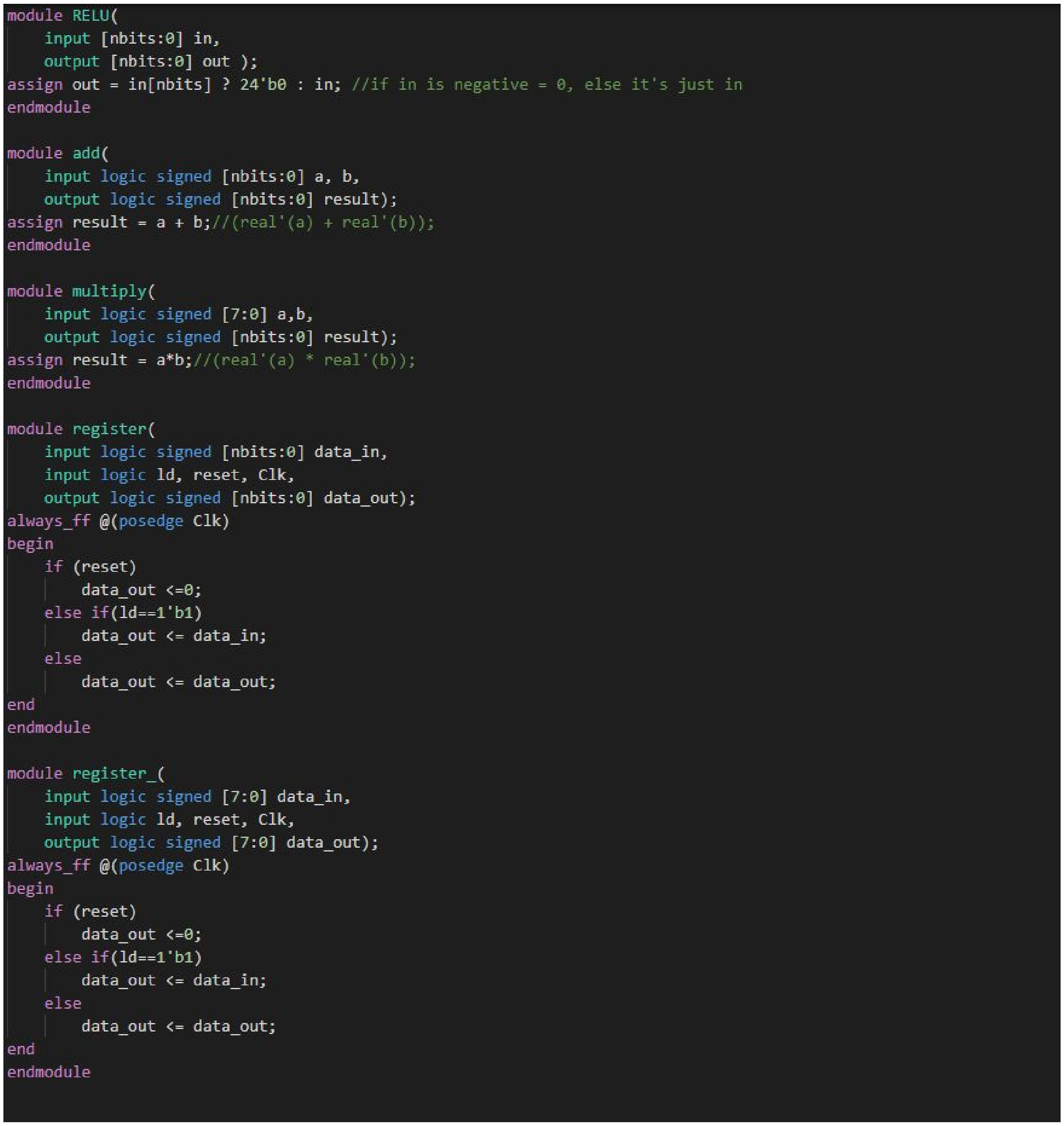


```python
class QConv2d(nn.Conv2d):

    def quantize_params(self):
        self.weight = nn.Parameter(quantize_uniform(self.weight, 8, 1, device='cpu'))
        self.bias = nn.Parameter(quantize_uniform(self.bias, 8, 1, device='cpu'))

    def forward(self, inputs):
        self.quantize_params()
        inputs = quantize_uniform(inputs, 8, 8, device='cpu')
        out = F.conv2d(inputs, self.weight, self.bias, self.stride, self.padding, self.dilation, self.groups)
        out = quantize_uniform(out, 8, 8, device='cpu')
        return out
```

Appendix B: The PE implementation in SystemVerilog

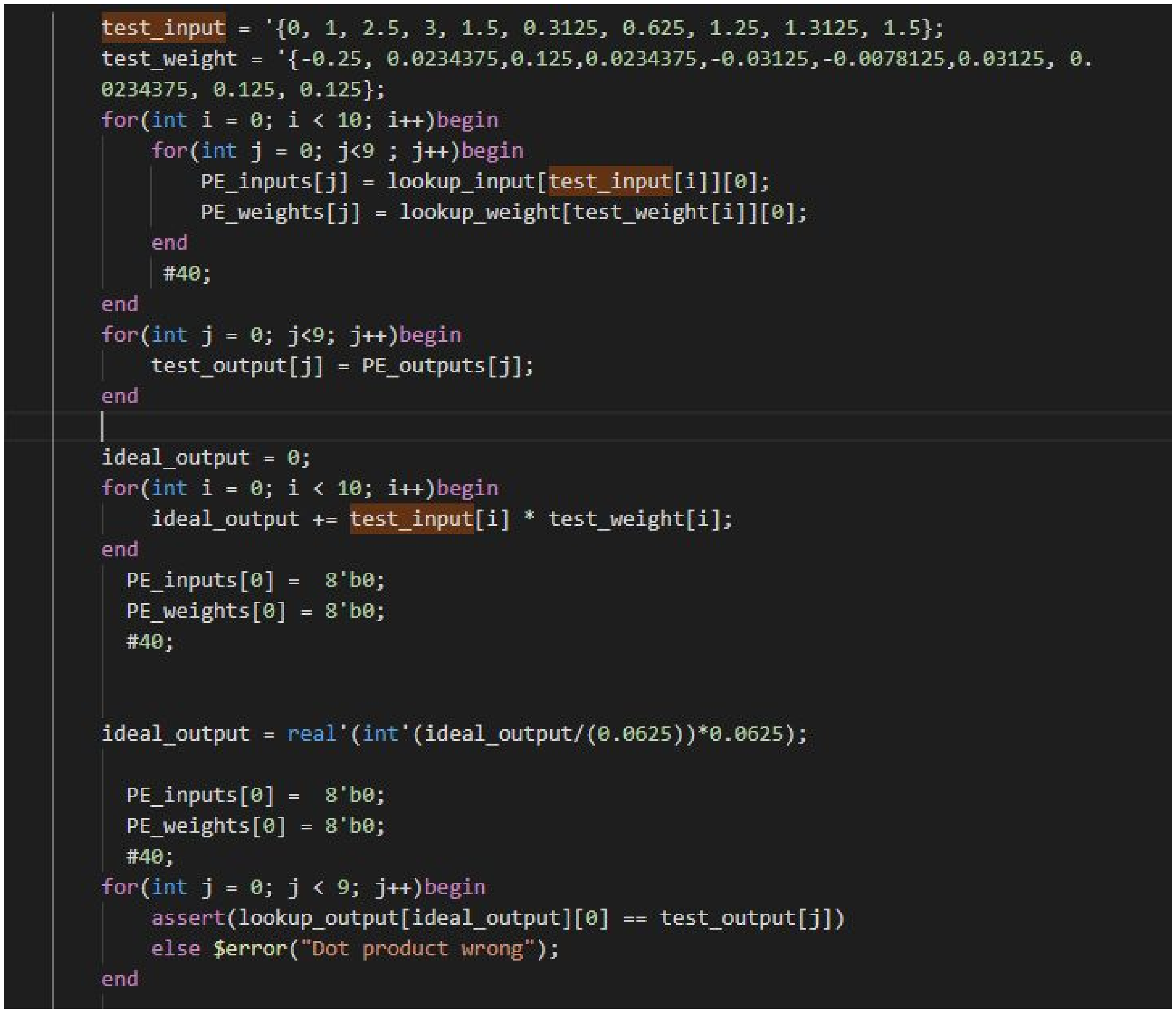


```
parameter nbits = 23 ; //note this is actually the precision in bits + 1 (includes zero)
//^note this is only in this file. make sure to change ram file and testbench if you change this value!
module fire(
    input logic reset, Clk,
    input logic ld_MAC, //these are control signals, currently i'm sending from testbench
    input logic signed [7:0] in_input[9],
    input logic signed [7:0] in_weight[9], //only used for 3x3 conv where all PEs make 1 pixel
    input logic signed [7:0] bias3x3,

    output logic signed [7:0] PE_added,
    output logic signed [7:0] out_PE[9]
    //if this module was a part of a large nn there could be signals here indicating where it is, when
);
logic signed [nbits:0] out_PE_[9];
logic signed [nbits:0] PE_added_;
always_comb begin
    for(int i = 0; i < 9; i++)begin
        if(out_PE_[i] != 24'b0)
        out_PE[i] = {out_PE_[i][23],out_PE_[i][13:7]} + out_PE_[i][6];//rounding up if the output is no
        else
        out_PE[i] = {out_PE_[i][23],out_PE_[i][13:7]};
    end
end
//here we should be doing 3*4 = 12
//PE PE1(.in_input(in_ram_out), .in_weight(w_ram_out), .out_activation(out_activation), .ld_MAC(ld_MAC)
PE PE_array[9](
    .in_input(in_input), //9 different inputs for each PE
    .in_weight(in_weight), //single weight broadcasted to all PEs
    .out_activation(out_PE_), //9 different outputs for each PE
    .ld_MAC(ld_MAC), //single ld signal broadcasted
    .*
);
logic signed out_bias_3x3;
multiply m_bias_3x3 (bias3x3, 8'd16 , out_bias_3x3);
assign PE_added_ = out_PE_[0] + out_PE_[1] + out_PE_[2] + out_PE_[3] + out_PE_[4] + out_PE_[5]
    + out_PE_[6] + out_PE_[7] + out_PE_[8] + out_bias_3x3;
assign PE_added = {PE_added_[23],PE_added_[13:7]+out_PE_[6]};

endmodule
module PE(
    input logic signed [7:0] in_input, in_weight,
    input logic Clk, reset, ld_MAC,
    output logic signed [nbits:0] out_activation);
logic signed [nbits:0] out_MAC, mult_out, add_out;
multiply m(in_weight, in_input, mult_out);
add a(mult_out, out_MAC, add_out);
register hold_MAC(.data_in(add_out), .data_out(out_MAC), .ld(ld_MAC), .*); //to initialize to zero hit
RELU activation(out_MAC, out_activation);
endmodule
```

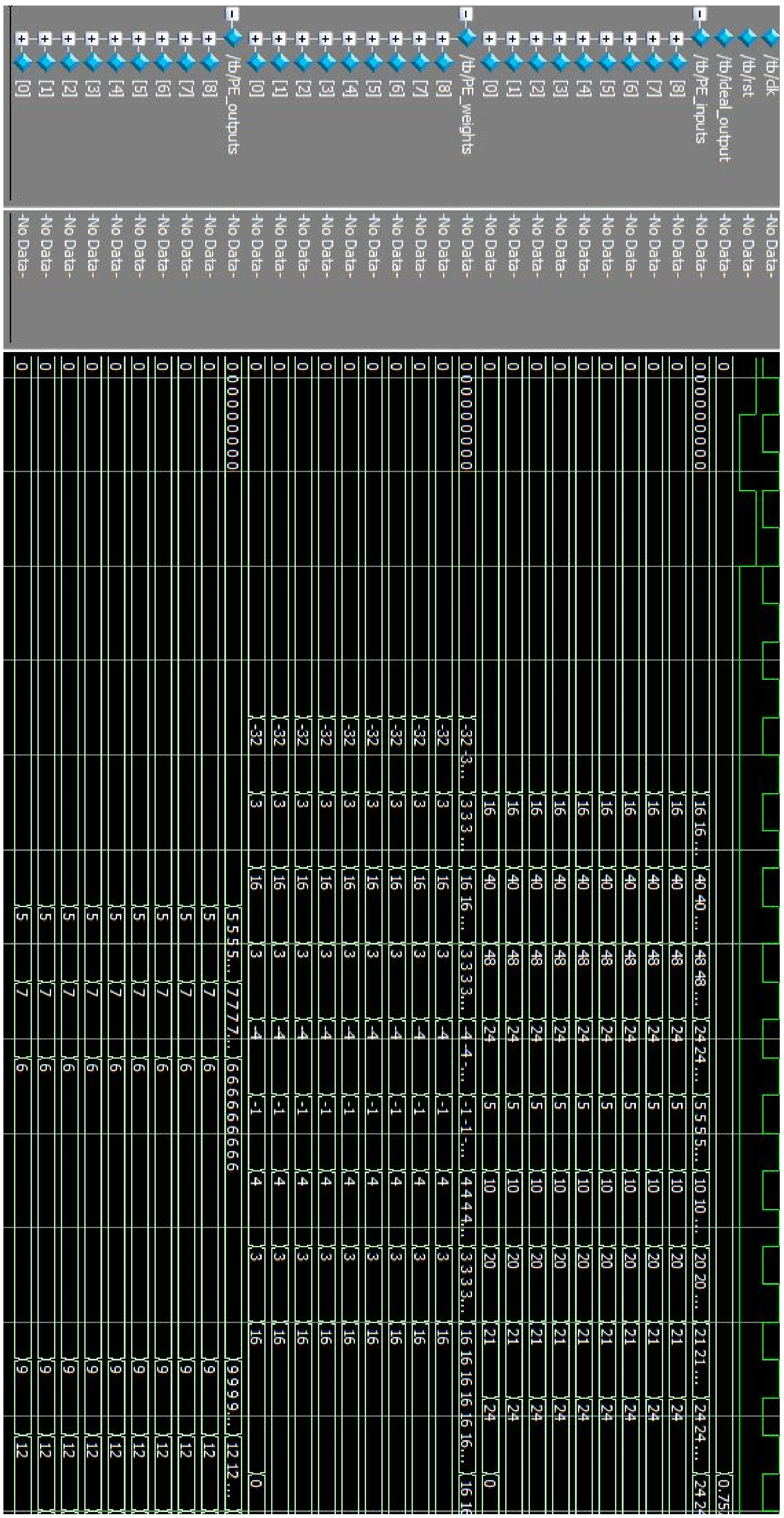


```
module RELU(
    input [nbits:0] in,
    output [nbits:0] out );
assign out = in[nbits] ? 24'b0 : in; //if in is negative = 0, else it's just in
endmodule

module add(
    input logic signed [nbits:0] a, b,
    output logic signed [nbits:0] result);
assign result = a + b;//(real'(a) + real'(b));
endmodule

module multiply(
    input logic signed [7:0] a,b,
    output logic signed [nbits:0] result);
assign result = a*b;//(real'(a) * real'(b));
endmodule

module register(
    input logic signed [nbits:0] data_in,
    input logic ld, reset, Clk,
    output logic signed [nbits:0] data_out);
always_ff @(posedge Clk)
begin
    if (reset)
        data_out <=0;
    else if(ld==1'b1)
        data_out <= data_in;
    else
        data_out <= data_out;
end
endmodule

module register_(
    input logic signed [7:0] data_in,
    input logic ld, reset, Clk,
    output logic signed [7:0] data_out);
always_ff @(posedge Clk)
begin
    if (reset)
        data_out <=0;
    else if(ld==1'b1)
        data_out <= data_in;
    else
        data_out <= data_out;
end
endmodule
```

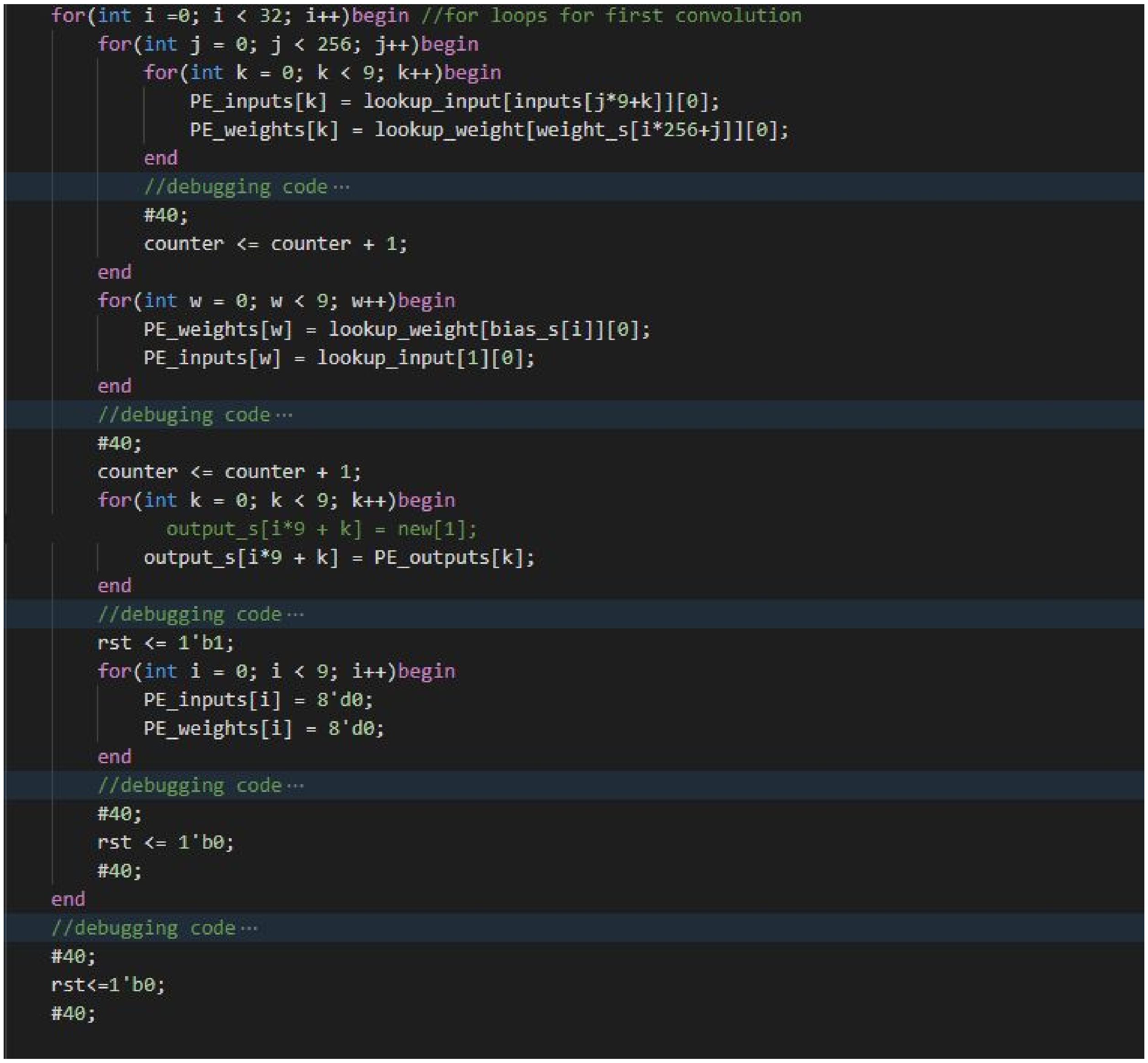
Appendix C: The Testbench Code to test functionality of the PEs



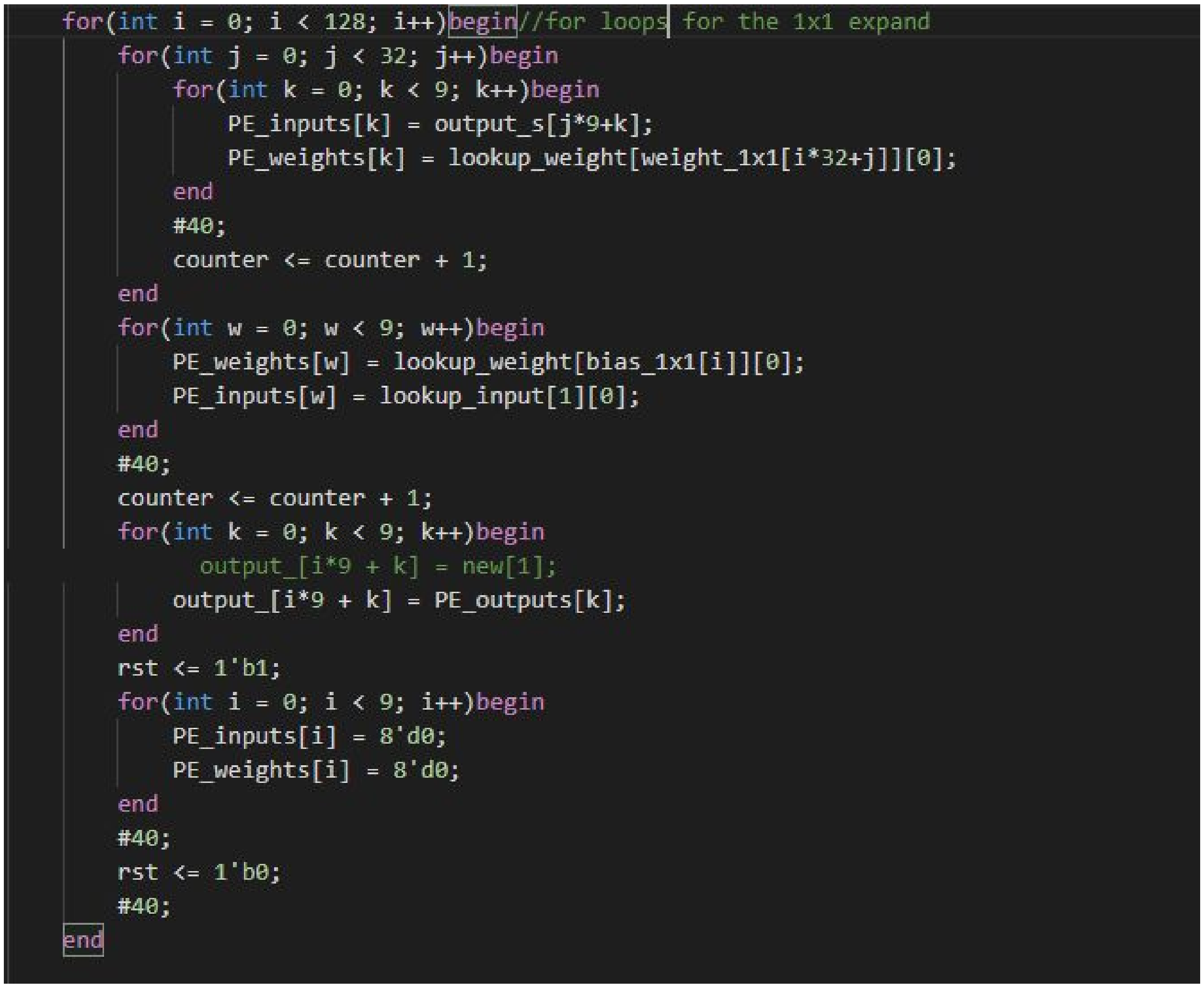
```
test_input = '{0, 1, 2.5, 3, 1.5, 0.3125, 0.625, 1.25, 1.3125, 1.5};
test_weight = '{-0.25, 0.0234375,0.125,0.0234375,-0.03125,-0.0078125,0.03125, 0.
0234375, 0.125, 0.125};
for(int i = 0; i < 10; i++)begin
    for(int j = 0; j<9 ; j++)begin
        PE_inputs[j] = lookup_input[test_input[i]][0];
        PE_weights[j] = lookup_weight[test_weight[i]][0];
    end
     #40;
end
for(int j = 0; j<9; j++)begin
    test_output[j] = PE_outputs[j];
end

ideal_output = 0;
for(int i = 0; i < 10; i++)begin
    ideal_output += test_input[i] * test_weight[i];
end
  PE_inputs[0] =  8'b0;
  PE_weights[0] = 8'b0;
  #40;


ideal_output = real'(int'(ideal_output/(0.0625))*0.0625);

  PE_inputs[0] =  8'b0;
  PE_weights[0] = 8'b0;
  #40;
for(int j = 0; j < 9; j++)begin
    assert(lookup_output[ideal_output][0] == test_output[j])
    else $error("Dot product wrong");
end
```

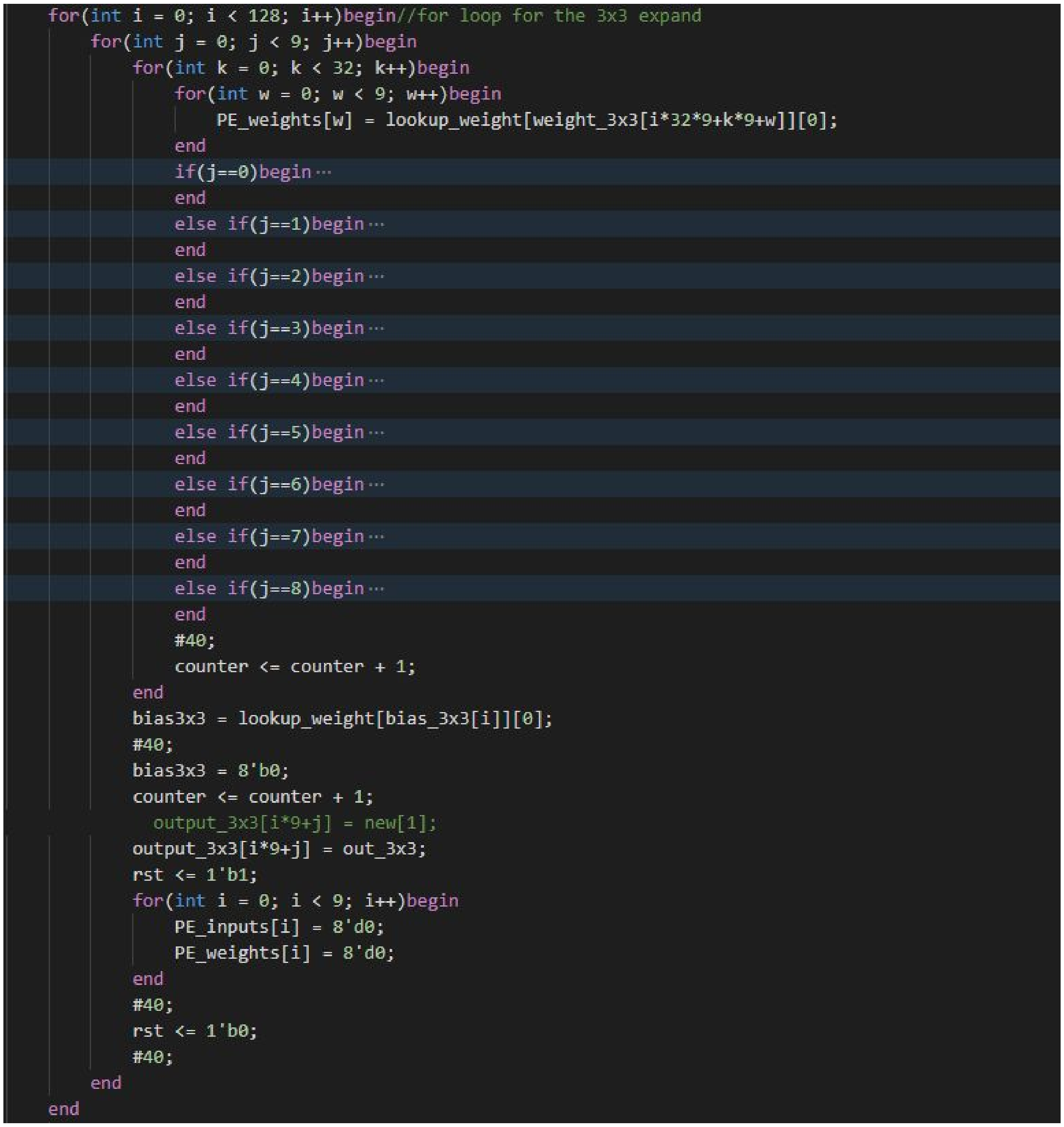

/tb/clk
/tb/rst
/tb/ideal_output
/tb/PE_inputs
[8]
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
/tb/PE_weights
[8]
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
/tb/PE_outputs
[8]
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
-No Data-

Appendix D: Testbench code for testing the layer with actual inputs

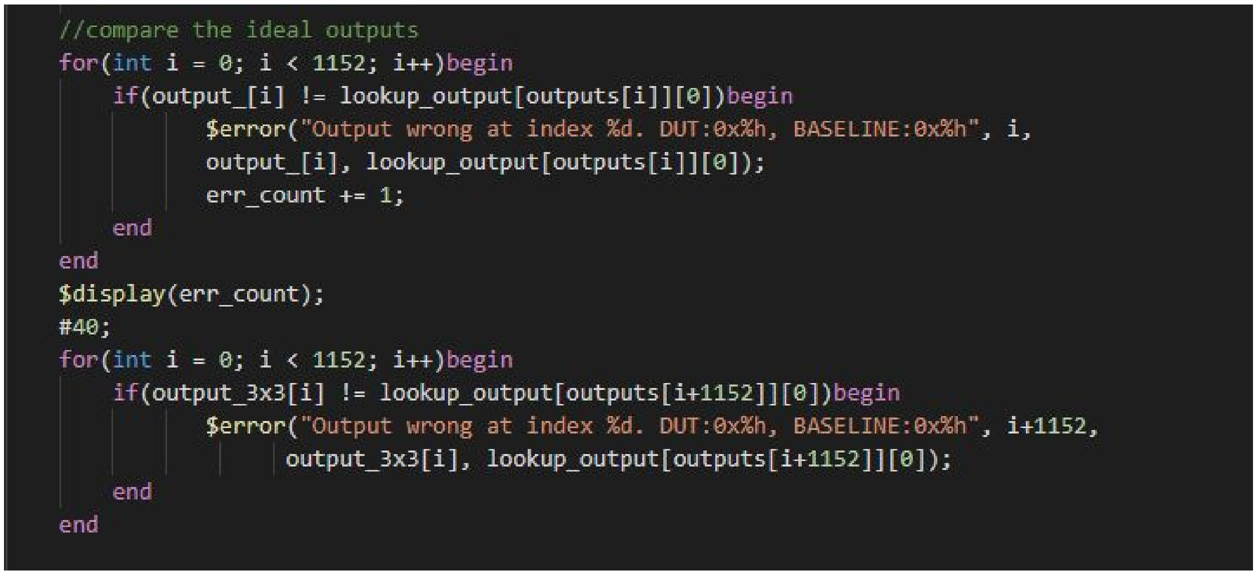


```
for(int i =0; i < 32; i++)begin //for loops for first convolution
    for(int j = 0; j < 256; j++)begin
        for(int k = 0; k < 9; k++)begin
            PE_inputs[k] = lookup_input[inputs[j*9+k]][0];
            PE_weights[k] = lookup_weight[weight_s[i*256+j]][0];
        end
        //debugging code···
        #40;
        counter <= counter + 1;
    end
    for(int w = 0; w < 9; w++)begin
        PE_weights[w] = lookup_weight[bias_s[i]][0];
        PE_inputs[w] = lookup_input[1][0];
    end
    //debuging code···
    #40;
    counter <= counter + 1;
    for(int k = 0; k < 9; k++)begin
          output_s[i*9 + k] = new[1];
        output_s[i*9 + k] = PE_outputs[k];
    end
    //debugging code···
    rst <= 1'b1;
    for(int i = 0; i < 9; i++)begin
        PE_inputs[i] = 8'd0;
        PE_weights[i] = 8'd0;
    end
    //debugging code···
    #40;
    rst <= 1'b0;
    #40;
end
//debugging code···
#40;
rst<=1'b0;
#40;
```


```
for(int i = 0; i < 128; i++)begin//for loops for the 1x1 expand
    for(int j = 0; j < 32; j++)begin
        for(int k = 0; k < 9; k++)begin
            PE_inputs[k] = output_s[j*9+k];
            PE_weights[k] = lookup_weight[weight_1x1[i*32+j]][0];
        end
        #40;
        counter <= counter + 1;
    end
    for(int w = 0; w < 9; w++)begin
        PE_weights[w] = lookup_weight[bias_1x1[i]][0];
        PE_inputs[w] = lookup_input[1][0];
    end
    #40;
    counter <= counter + 1;
    for(int k = 0; k < 9; k++)begin
          output_[i*9 + k] = new[1];
        output_[i*9 + k] = PE_outputs[k];
    end
    rst <= 1'b1;
    for(int i = 0; i < 9; i++)begin
        PE_inputs[i] = 8'd0;
        PE_weights[i] = 8'd0;
    end
    #40;
    rst <= 1'b0;
    #40;
end
```



```
for(int i = 0; i < 128; i++)begin//for loop for the 3x3 expand
    for(int j = 0; j < 9; j++)begin
        for(int k = 0; k < 32; k++)begin
            for(int w = 0; w < 9; w++)begin
                PE_weights[w] = lookup_weight[weight_3x3[i*32*9+k*9+w]][0];
            end
            if(j==0)begin···
            end
            else if(j==1)begin···
            end
            else if(j==2)begin···
            end
            else if(j==3)begin···
            end
            else if(j==4)begin···
            end
            else if(j==5)begin···
            end
            else if(j==6)begin···
            end
            else if(j==7)begin···
            end
            else if(j==8)begin···
            end
            #40;
            counter <= counter + 1;
        end
        bias3x3 = lookup_weight[bias_3x3[i]][0];
        #40;
        bias3x3 = 8'b0;
        counter <= counter + 1;
          output_3x3[i*9+j] = new[1];
        output_3x3[i*9+j] = out_3x3;
        rst <= 1'b1;
        for(int i = 0; i < 9; i++)begin
            PE_inputs[i] = 8'd0;
            PE_weights[i] = 8'd0;
        end
        #40;
        rst <= 1'b0;
        #40;
    end
end
```



```
//compare the ideal outputs
for(int i = 0; i < 1152; i++)begin
    if(output_[i] != lookup_output[outputs[i]][0])begin
            $error("Output wrong at index %d. DUT:0x%h, BASELINE:0x%h", i,
            output_[i], lookup_output[outputs[i]][0]);
            err_count += 1;
    end
end
$display(err_count);
#40;
for(int i = 0; i < 1152; i++)begin
    if(output_3x3[i] != lookup_output[outputs[i+1152]][0])begin
            $error("Output wrong at index %d. DUT:0x%h, BASELINE:0x%h", i+1152,
                  output_3x3[i], lookup_output[outputs[i+1152]][0]);
    end
end
```