FPGram

Project Report

Tonye Brown, tb2553
Diana Valverde-Paniagua, drv2110

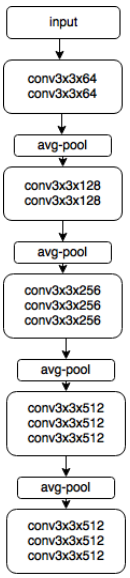
Contents

1. Introduction

Convolutional neural networks are used to apply artificial neural network functions on 2D image data. The network is composed of an input layer which takes in a matrix to be processed, hidden layers where kernels, the weights at each neuron, are applied to the image as convolutions, and fully connected layers that compute the output of the network. The filters used as kernels are used to emphasize certain geometric shapes and textures from which an attribute about the input image can be inferred. These networks are more suited for working with images than regular feed-forward neural network as less weights can be used to process the image therefore the network is more constrained. Certain aspects of this computational model make it suitable for hardware acceleration. The convolution and pooling processes are easily parallelized when each output pixel is considered separately. Hardware implementation of the back propagation would also speed up image processing.

Background

We adapted a model of a convolutional neural network based on the 19-layer Very Deep Convolutional Neural Network by Simonyan and Zisserman at Oxford University.



The network takes in an 224 by 224 matrix consisting of 3 channels. This image goes through 5 stages of convolution and average pooling. Our model is taken from the adaptation of the VGG network used in the research paper, A Neural Network of Artistic Style. This paper describs how convolutional neural networks can be used to create an image that has the content of one image and the style of another. This is based on the theory that recognition of style and recognition of content are separate, and that if we minimize the distance between both features on a white noise image, we are able to create an image with a specified style and content. The back-propation performed on

this network is not on the weights of the neurons but rather on the modifies the white noise image in order to match the expected output based on the content image and the style image Our project aimed to implement this network in hardware. Overall, the lab can be split into two components: - Memory Controller - VGG Network functions

The computations performed at each network at each network increases exponentially as the number of channels at each layer doubles as it moves from one stage to another. Therefore in addition to using the DDR3 memory on the FPGA, a protocol for reading the image data into buffers from DDR3 and vice versa and had to be implemented, in order to limit the amount of time spent sending data across busses.

Related Work

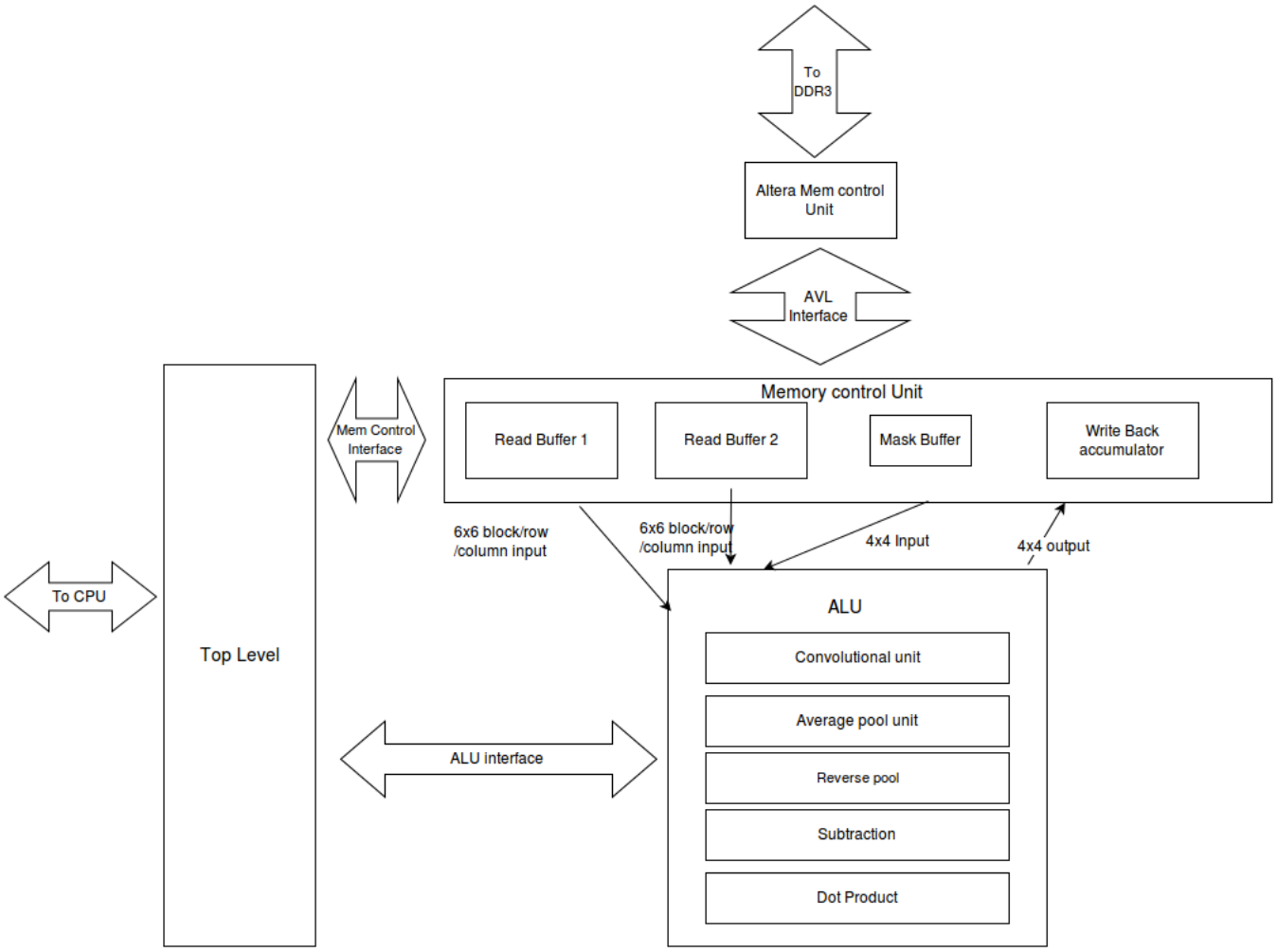
This project was based off of the paper *A Neural Network of Artistic Style* found at `http://arxiv.org/pdf/1508.06576v2.pdf`.

Goals

Initially this project aimed to implement a fully working system where a user would take in image or input an image previously downloaded, choose from an assortment of preset style images and an output image would appear on the VGA screen. Unfortunately, due to unforseen limitations of the board, the project had to be scaled back.

2. Design

Block Diagram



3. ISA

The instruction set for this processor is one 64-bit word, that can be broken down into two 32-bit segments. Memory instructions and ALU instructions are distinguished through the first bit, which is high if it is a memory instruction and low if it is an ALU instruction. The upper 32 bits of a memory instruction is composed of 1 bit reset, 1 bit pad (leaves a pad of 0's along the edges of the input channel), 26 bit instruction address and 4 bit buffer target. The lower 32 bits denote 8 bits for the number of rows in the channel, 8 bits denote the number of columns, 8 bits denote the stride for cases with strided memory access, and 8 bits are left empty.

The upper 32 bits of ALU instructions are composed of 3 bits for the ALU operation, 16 bits for the sub index in the accumulation buffer (used in Gram matrix calculations), 2 bits for the sub block of the input or output buffer (used in average pooling), 1 bit for reversing the mask (used in convolution), and 1 bit to denote whether the ALU access is done in blocks or by columns (blocks are used for most units, but for Gram matrix and artistic style loss calculations, the memory accesses are done in "columns" of the input buffer. The lower 32 bits are used for calculating which cells in the RAMs to fetch, because of of the unorthodox ordering of the memory in the blocks (see table 3.1). This aspect of the ISA trades instruction complexity for processor efficiency, because by staggering indices in the RAMs, all blocks, rows and columns can be accessed in one clock cycle simultaneously.

Memory instruction decoding

```
assign inst_op = instruction[63:60];

//Memory instruction decoding
logic        inst_reset;
logic        inst_pad;
logic [25:0] inst_addr;
logic  [7:0] zzz;
logic  [7:0] inst_rows;
logic  [7:0] inst_cols;
logic  [7:0] inst_stride;

assign {inst_reset, inst_pad, inst_addr, zzz, inst_rows,
↪ inst_cols, inst_stride} = instruction[57:0];
```

Table 3.1: RAM - cells laid out in DDR3 memory order, numbers are the layer within the rows of RAM that contain the data

0	1	2	3	12	13	...
4	5	6	7	0	1	...
8	9	10	11	4	5	...
12	13	14	15	8	9	...
3	0	1	2	13		
...	...	...	...			

ALU instruction decoding

```
//ALU instruction decoding
logic [2:0]  inst_alu_op;
logic [15:0] inst_sub_index;
logic [1:0]  inst_sub_block;
logic        inst_rev_mask;
logic        inst_block_in_col_modde;
logic [7:0]  inst_block_in_row;
logic [7:0]  inst_block_in_col;
logic [7:0]  inst_block_out_row;
logic [7:0]  inst_block_out_col;

assign inst_alu_op = instruction[58:56];

assign inst_block_in_col_mode = instruction[48];
assign inst_sub_index = instruction[47:32];
assign inst_sub_block = instruction[33:32];
assign inst_rev_mask  = instruction[32];

assign inst_block_in_row  = instruction[31:24];
assign inst_block_in_col  = instruction[23:16];
assign inst_block_out_row = instruction[15:8];
assign inst_block_out_col = instruction[7:0];

assign alu_i.operation = inst_alu_op;
assign alu_i.sub_index = inst_sub_index;
assign alu_i.sub_block = inst_sub_block;
assign alu_i.rev_mask  = inst_rev_mask;

assign read_buff1.col_mode  = inst_block_in_col_mode;
assign read_buff2.col_mode  = inst_block_in_col_mode;

assign read_buff1.block_row  = inst_block_in_row;
assign read_buff2.block_row  = inst_block_in_row;
assign accumulator.block_row = inst_block_out_row;

assign read_buff1.block_col  = inst_block_in_col;
assign read_buff2.block_col  = inst_block_in_col;
assign accumulator.block_col = inst_block_out_col;
```

4. Memory Control Unit

An Altera memory control unit was created using a pre-written IP component from the manufacturer. From this component a read and write module was written to interface with the external memory.

Read Buffer I

The first read buffer is 256 by 256 matrix buffer of 27 bits. It is composed of 64 RAMs so that data can accessed in one clock cycle.This is the main buffer used. It reads from the DDR3 through the AVL interface based on the stride, row and block inputs provided through the intstruction set.

Read Buffer II

The second read buffer is a 256 by 128 buffer of 27 bits that reads from the DDR3.This is a secondary buffer only used for the gram matrix function performed on the white noise image.

Mask Buffer

4 by 4 buffer of 8 bits used to load the weight that are to be used by the ALU units. This is controlled through top level device as weights are stored on the DDR3 memory.

Write Back Accumulator

Another 256 by 256 buffer now composed of 16 dual-port RAMs. At the end of each layer, the results of the convolutional layers has to be written across the channels or across kernels applied during the backpropagation. The write-back accumulator makes this process more efficient.

5. ALU

Convolutional Unit

The convolutional unit instantiates several smaller units to perform perform convolution on a 6x6 to 4x4 27-bit pixel block in parallel. A channel of the entire image fed through this unit from the read unit and the mask is taken from the mask buffer of the memory control unit.

Average Pooling Unit

A hardware implementation of the average pooling function of the neural network. The 4x4 block matrix of image data is pooled into a 2x2 matrix.

Reverse Pool

A reverse pooling unit used for the backwards pass. It maps a 2x2 matrix to a 4x4 matrix.

Subtraction

This module implements a 16-bit vector subtraction. It it is used in backpropagation for calculating content loss and its derivative.

Dot Product

A dot product unit for the backwards pass. The unit performs fixed point dot product multiplication on the matrices.

6. Lessons Learned

We underestimated the time and resources that this project would take. As previously stated, we first assumed that it would be possible to only implement parts of the VGG network on the board and control it from the software as the entire network could not be fit on the board. However, we realized that that sending data back and and forth between layers of the network would make the system too slow and any acceleration we would get from hardware inconsequential. Contolling the image data from instructions sent over the CPU led to a lot of issues with timing, reading and writing the appropriate block of data and fixed point logic that we had not anticipated would take so much of the development time.

Diana

Hardware

One of the main principles I've learned from this class has been the importance of reading documentation straight from the manufacturer. As an intermediate-level programmer, I've looked at the *Java* or *C* reference manuals a couple times to look for answers to very specific questions, but, perhaps appallingly, skimming over answers on stack overflow has been sufficient for the majority of things I've done. This is because programming is very abstract and modular in software - if you see an example of a program in C, you'll get a general idea of what it looks like in any other programming language. For large projects in hardware, however, this is simply not the case. There's absolutely no getting around understanding the minutiae of the implementation of what you're doing *and* what compiler is doing. Some problems you will run across are a very specific combination of problems from the manufacturer of the board, the IP components you're using, the compilation settings you've selected and the logic you're writing. It can be hard to know the difference between all of them unless you really take the time to understand what is going on. As someone who hadn't taken any other hardware-related class prior to this one, let's just say there was a steep learning curve. Still, I think I came out with a great appreciation for hardware and interest in doing some projects in my own time.

Miscellaneous

This is one of the few classes at Columbia that actually treats you like an adult. That can be a good thing or a bad thing.

Tonye

Hardware

A big takeaway from this class for me is the importance of starting early and more specifically, testing early. The documentation for things such as communicating with the DDR3 or testing large systems with ModelSim was dense and often times not helpful for what we were trying to do, so often times you have o hammer things out by yourself and that means failing fast and failing often. Most of this project was spent rewriting the memory interface or the function units because there was something that we hadn't taken into account before. Similarly, I think in hardware projects you definitely have to be on the same project with everyone in the group because a small change in one unit can force you to rewrite the entire system. Whereas software can be more generic and portable, hardware requires constant communication to ensure effeciency.

Miscellaneous

There are a lot of problems that may be specific to your board. For example, a problem with the HPS made us unable to boot the sof file and load the Linux. It is hard to gauge whether it's a problem with what you've done in configuring the board or the way you've written the images onto the memory. This class is a great introduction to Embedded Systems.I think anybody interested in the field will be able to get a good sense of what it entails by taking this class.

7. Code Listing

ALU.sv

```
/********* ALU OPERATION DESCRIPTIONS AND ENCODING **********
*  Three bits give the operation ID for the ALU to execute.
*     000: convolution          FLAGS: rev_mask high
*    001: forward pool         FLAGS: sub_block
*    010: backward pool       FLAGS: sub_block
*    011: reLU
*    100: difference
*    101: dot product         FLAGS: sub_index
*    110: backprop reLu
*
*   Summary of ALU usage in the neural network:
*
*  FORWARD PASS
*    - convolution
*    - reLU
*    - forward pool with the sub_block flag
*  LOSS FUNCTIONS
*    Content loss:
*      - difference
*      - dot product
*    Artistic loss:
*      - Gram matrix: dot product with the sub_index flag
*       - Total loss:
*        - difference
*        - dot product
*  DERIVATIVES:
*    Content loss:
*      - difference
*    Artistic loss:
*      - matrix Multiplication: Dot product
*      - difference
*     Backwards pass:
*      - backward pool         FLAGS: sub_block
*      - backprop reLu
*      - dot product
*      - convolution           FLAGS: rev_mask low
*/

module ALU (
  input iCLK,
  READ_BUFFER.ALU in_block1,
```

```
  READ_BUFFER.ALU in_block2,
  MASK_BUFFER.ALU mask,
  WRITE_BACK.ALU out_block,
  ALU_i.ALU from_top
);
parameter PADDED_SIZE = 6;
parameter NOT_PADDED_SIZE = 4;
parameter NOT_PADDED_BLOCK = NOT_PADDED_SIZE * NOT_PADDED_SIZE;

parameter WORD_SIZE = 27;

// 8x8 inputs
reg signed [WORD_SIZE-1:0] no_pad1 [15:0];
reg signed [WORD_SIZE-1:0] no_pad2 [15:0];
genvar i,j;
generate for (i = 0; i < NOT_PADDED_SIZE; i++) begin: for_i
  for (j = 0; j < NOT_PADDED_SIZE; j++) begin: for_j
    assign no_pad1[i+j*NOT_PADDED_SIZE] =
 ↪  in_block1.block[i+j*PADDED_SIZE];
    assign no_pad2[i+j*NOT_PADDED_SIZE] =
 ↪  in_block2.block[i+j*PADDED_SIZE];
  end
end
endgenerate



//Instantiate arithmetic modules
reg signed [WORD_SIZE-1:0] ap_out [NOT_PADDED_BLOCK-1:0];
avg_pool #(.WORD_SIZE(WORD_SIZE)) av (
  .pixels_in (no_pad1),
  .pixels_out (ap_out),
  .sub_block (from_top.sub_block)
);

reg signed [WORD_SIZE-1:0] diff_out [NOT_PADDED_BLOCK-1:0];
difference #(.WORD_SIZE(WORD_SIZE)) d (
  .in1 (no_pad1),
  .in2 (no_pad2),
  .out (diff_out)
);

reg signed [WORD_SIZE-1:0] bp_out [NOT_PADDED_BLOCK-1:0];
backprop_pool #(.WORD_SIZE(WORD_SIZE)) bp (
  .derivative_in (no_pad1),
  .derivative_out (bp_out),
  .sub_block (from_top.sub_block)
);

reg signed [WORD_SIZE-1:0] br_out [NOT_PADDED_BLOCK-1:0];
backprop_relu #(.WORD_SIZE(WORD_SIZE)) br (
  .derivative_in (no_pad1),
  .pixels (no_pad2),
  .derivative_out (br_out)
```

```
);

reg signed [WORD_SIZE-1:0] rm_reversed [8:0];
reverse_mask #(.WORD_SIZE(WORD_SIZE)) rm (
  .kernel (mask.mask),
  .reversed (rm_reversed)
);

reg signed [WORD_SIZE-1:0] convo_mask [8:0];
reg signed [WORD_SIZE-1:0] convo_out [NOT_PADDED_BLOCK-1:0];
reg dp_mode;
assign convo_mask = from_top.rev_mask ? rm_reversed : mask.mask;
convolution #(.WORD_SIZE(WORD_SIZE)) convo(
  .pixels_in (in_block1.block),
  .mask (convo_mask),
  .pixels_out (convo_out)
);

reg signed [WORD_SIZE-1:0] dp_out [NOT_PADDED_BLOCK-1:0];
dot_product #(.WORD_SIZE(WORD_SIZE)) d_prod(
      .out_index(from_top.sub_index),
      .in1(no_pad1),
      .in2(no_pad2),
      .out(dp_out)
);

reg signed [WORD_SIZE-1:0] r_pixels_out [NOT_PADDED_BLOCK-1:0];
relu #(.WORD_SIZE(WORD_SIZE)) r (
  .pixels_in (no_pad1),
  .pixels_out (r_pixels_out)
);


logic [WORD_SIZE-1:0] out_block_buff [NOT_PADDED_BLOCK-1:0];
reg [1:0] state;
initial begin state <= 0;
  from_top.ready <= 1;
end

// 0: Ready for instructions / Executing instruction
// 1: Write/Accumulate
always @(posedge iCLK)
begin
  dp_mode='x;
  case (state)
   0 : begin
      if (from_top.execute) begin
        from_top.ready <= 0;

        state <= 1;
        case (from_top.operation)
          3'b000 : begin
                  dp_mode=0;
                  out_block_buff <= convo_out;
```

```
148                 end
149             3'b001 : out_block_buff <= ap_out;
150             3'b010 : out_block_buff <= bp_out;
151             3'b011 : out_block_buff <= r_pixels_out;
152             3'b100 : out_block_buff <= diff_out;
153             3'b101 :  begin
154                     dp_mode=1;
155                     out_block_buff <= convo_out;
156                 end
157             3'b110 : out_block_buff <= br_out;
158           endcase
159         end
160       end
161     1 : begin //Allow register retiming to pipeline the ALU to two
    ↪  cycles
162       out_block.block <= out_block_buff;
163       out_block.accumulate <= 1;
164       state <= 2;
165     end
166     2 : begin
167         out_block.accumulate <= 0;
168         state <= 0;
169         from_top.ready <= 1;
170       end
171   endcase
172 end
173 endmodule
```

avg_pool.sv

```
1  module avg_pool(pixels_in, pixels_out, sub_block);
2
3  parameter WIDTH_IN = 4;
4  parameter WIDTH_OUT = 2;
5
6  parameter WORD_SIZE = 27;
7  input signed [WORD_SIZE-1:0] pixels_in [(WIDTH_IN*WIDTH_IN - 1):0];
8  input [1:0] sub_block;
9  output signed [WORD_SIZE-1:0] pixels_out [(WIDTH_IN*WIDTH_IN - 1):0];
10
11 logic signed [WORD_SIZE-1:0] pooled_pix [WIDTH_OUT*WIDTH_OUT-1:0];
12 genvar i,j;
13 generate for (i = 0; i < WIDTH_OUT; i++) begin: for_i
14   for (j = 0; j < WIDTH_OUT; j++) begin: for_j
15     assign pooled_pix[(i + j*WIDTH_OUT)] = (pixels_in[(i*2 +
   ↪  j*2*WIDTH_IN)]
16                                 + pixels_in[(i*2 + j*2*WIDTH_IN + 1)]
17                                 + pixels_in[(i*2 + j*2*WIDTH_IN +
   ↪  WIDTH_IN)]
18                                 + pixels_in[(i*2 + j*2*WIDTH_IN +
   ↪  WIDTH_IN + 1)]) >>> $signed(2);
19   end
20 end
21 endgenerate
22
23 always_comb begin
24   for (int i = 0; i < WIDTH_IN*WIDTH_IN; i++) pixels_out[i] = 0;
25   case(sub_block)
26   0: begin
27     pixels_out[0*WIDTH_IN +: WIDTH_OUT] = pooled_pix[0*WIDTH_OUT
   ↪  +:WIDTH_OUT];
28     pixels_out[1*WIDTH_IN +: WIDTH_OUT] = pooled_pix[1*WIDTH_OUT
   ↪  +:WIDTH_OUT];
29     end
30   1: begin
31     pixels_out[0*WIDTH_IN + WIDTH_OUT +: WIDTH_OUT] =
   ↪  pooled_pix[0*WIDTH_OUT +:WIDTH_OUT];
32     pixels_out[1*WIDTH_IN + WIDTH_OUT +: WIDTH_OUT] =
   ↪  pooled_pix[1*WIDTH_OUT +:WIDTH_OUT];
33     end
34   2: begin
35     pixels_out[2*WIDTH_IN +: WIDTH_OUT] = pooled_pix[0*WIDTH_OUT
   ↪  +:WIDTH_OUT];
36     pixels_out[3*WIDTH_IN +: WIDTH_OUT] = pooled_pix[1*WIDTH_OUT
   ↪  +:WIDTH_OUT];
37     end
38   3: begin
39     pixels_out[2*WIDTH_IN + WIDTH_OUT +: WIDTH_OUT] =
   ↪  pooled_pix[0*WIDTH_OUT +:WIDTH_OUT];
```

```
40      pixels_out[3*WIDTH_IN + WIDTH_OUT +: WIDTH_OUT] =
    ↪   pooled_pix[1*WIDTH_OUT +:WIDTH_OUT];
41      end
42    endcase
43  end
44
45
46  endmodule
```

backprop_relu.sv

```
1  module backprop_relu(
2    derivative_in,
3    pixels,
4    derivative_out);
5
6  parameter SIZE = 16;
7
8  parameter WORD_SIZE = 27;
9
10 input signed [WORD_SIZE-1:0] derivative_in [(SIZE - 1):0];
11 input signed [WORD_SIZE-1:0] pixels [(SIZE - 1):0];
12 output signed [WORD_SIZE-1:0] derivative_out [(SIZE - 1):0];
13
14 genvar i;
15 generate for (i = 0; i < SIZE; i++) begin : for_i
16   assign derivative_out[i] = (pixels[i] < $signed(0))? 1'b0 :
   ↪  derivative_in[i];
17 end
18 endgenerate
19
20 endmodule
```

conv.sv

```
module conv
  #(parameter WORD_SIZE=27)
  (
  input [8:0][WORD_SIZE-1:0] in ,
  input signed[WORD_SIZE-1:0] kernel [8:0],
  output signed [WORD_SIZE-1:0] out);
parameter DSP_MULT_COUNT = 9;

logic signed [39:0] intermediate;
logic signed [39:0] prods [8:0];

always_comb begin
  for(integer i=0; i<DSP_MULT_COUNT; i++) prods[i] = $signed(in[i]) *
  ↪  $signed(kernel[i]);
end
genvar j;
generate
  for(j=DSP_MULT_COUNT; j<9; j++) begin : for_j
    soft_lpmmult mult(
          .dataa($signed(in[j])),
          .datab(kernel[j]),
          .result(prods[j])
          );
  end
endgenerate

assign intermediate =  prods[0] + prods[1] + prods[2]
              + prods[3] + prods[4] + prods[5]
              + prods[6] + prods[7] + prods[8];

assign out = intermediate >>> $signed(13);
endmodule
```

convolution.sv

```
module convolution(
  pixels_in,
  mask,
  pixels_out);


parameter WIDTH_IN = 6;
parameter WIDTH_OUT = WIDTH_IN - 2;

parameter WORD_SIZE = 27;

input signed [WORD_SIZE-1:0] pixels_in [(WIDTH_IN * WIDTH_IN - 1):0];
input signed [WORD_SIZE-1:0] mask [8:0];
output signed [WORD_SIZE-1:0] pixels_out [(WIDTH_OUT * WIDTH_OUT - 1):0];


logic signed [WORD_SIZE-1:0] pix_out [(WIDTH_OUT * WIDTH_OUT - 1):0];

genvar i,j;
generate for (i = 1; i < (WIDTH_IN - 1); i++) begin: for_i
  for (j = 1; j < (WIDTH_IN - 1); j++) begin: for_j
    logic signed [WORD_SIZE-1:0] mask_tmp [8:0];

    localparam NUM_DSP = (j>3)?5:6;

    always_comb begin
        mask_tmp = mask;
    end

    conv #(NUM_DSP) con
      (
      .in  ({pixels_in[((i + (j*WIDTH_IN))-(WIDTH_IN - 1))], pixels_in[((i + (j*WIDTH_IN))-(WIDTH_IN))], pixels_in[((i + (j*WIDTH_IN))-(WIDTH_IN + 1))],
            pixels_in[((i + (j*WIDTH_IN))+ 1)], pixels_in[((i + (j*WIDTH_IN)))],           pixels_in[((i + (j*WIDTH_IN))-1)],
            pixels_in[((i + (j*WIDTH_IN))+(WIDTH_IN + 1))], pixels_in[((i + (j*WIDTH_IN))+(WIDTH_IN))], pixels_in[((i + (j*WIDTH_IN))+(WIDTH_IN - 1))]
          }),
      .kernel (mask_tmp),
      .out (pix_out[(i-1)+(j-1)*WIDTH_OUT]));
    assign pixels_out[(i-1)+(j-1)*WIDTH_OUT] = pix_out[(i-1)+(j-1)*WIDTH_OUT];
  end
end

endgenerate
```

```
43
44
45 endmodule
```

difference.sv

```
module difference(
  in1,
  in2,
  out);

parameter SIZE = 16;
parameter WORD_SIZE = 27;

input signed [WORD_SIZE-1:0] in1 [(SIZE-1):0];
input signed [WORD_SIZE-1:0] in2 [(SIZE-1):0];
output signed [WORD_SIZE-1:0] out [(SIZE-1):0];

genvar i;
generate for (i = 0; i < SIZE; i++) begin : for_i
  assign out[i] = in1[i] - in2[i];
end
endgenerate

endmodule
```

dot_product.sv

```
module dot_product(
  out_index,
  in1,
  in2,
  out);

parameter SIZE = 16;

parameter WORD_SIZE = 27;

input signed [WORD_SIZE-1:0] out_index;
input signed [WORD_SIZE-1:0] in1 [(SIZE-1):0];
input signed [WORD_SIZE-1:0] in2 [(SIZE-1):0];
output signed [WORD_SIZE-1:0] out [(SIZE-1):0];

logic signed [WORD_SIZE-1:0] prod [(SIZE-1):0];
genvar i;
generate for (i = 0; i < SIZE; i++) begin : for_i
  logic[39:0] tmp;
  soft_lpmmult mult(
          .dataa(in1[i]),
          .datab(in1[i]),
          .result(tmp)
          );
  assign prod[i] = tmp >>> 13;
end
endgenerate

integer s;
reg signed [WORD_SIZE-1:0] sum;

assign sum = prod.sum();

always_comb
  begin
  for(s=0; s<SIZE; s++) out[s]=0;
  out[out_index]=sum;
end

endmodule
```

FatBuffer.sv

```
module FatBuffer (
input clk,
input init,
output init_busy,

input read_block_en,
input read_col_as_block_en,
input write_block_en,
input [6:0] block_row,
input [6:0] block_col,
input signed [WORD_SIZE-1:0] block_data_in [USR_BLOCK_SIZE-1:0],
output signed [WORD_SIZE-1:0] block_data_out [USR_BLOCK_SIZE-1:0],

input read_vword_en,
input write_vword_en,
input [9:0] vword_row,
input [9:0] vword_col,
input [127:0] vword_data_in,
output [127:0] vword_data_out

);

parameter WORD_SIZE = 27;

parameter BUFFER_WIDTH = 128;
parameter ADDR_WIDTH = 8;

parameter MEM_BLOCK_WIDTH = 8;
parameter MEM_BLOCK_SIZE = MEM_BLOCK_WIDTH * MEM_BLOCK_WIDTH;

parameter MEM_BLOCKS_PER_ROW = BUFFER_WIDTH / MEM_BLOCK_WIDTH;


parameter USR_BLOCK_WIDTH = 6;
parameter USR_BLOCK_SIZE = USR_BLOCK_WIDTH * USR_BLOCK_WIDTH;

parameter USR_BLOCK_SPACING = 4;

parameter WRITE_DISABLE = 0;
parameter READ_BLOCK_DISABLE = 0;
parameter READ_COL_DISABLE = 0;
parameter READ_DISABLE = READ_BLOCK_DISABLE && READ_COL_DISABLE;

logic [ADDR_WIDTH-1:0] address_a [MEM_BLOCK_SIZE-1:0];
logic [WORD_SIZE-1:0] data_a [MEM_BLOCK_SIZE-1:0];
logic wren_a [MEM_BLOCK_SIZE-1:0];
logic [WORD_SIZE-1:0] q_a [MEM_BLOCK_SIZE-1:0];

logic [WORD_SIZE-1:0] init_dataout;
logic [ADDR_WIDTH-1:0] init_address;
```

```
51  logic init_wren;
52
53  logic [ADDR_WIDTH-1:0] address_b [MEM_BLOCK_SIZE-1:0];
54  logic [WORD_SIZE-1:0] data_b [MEM_BLOCK_SIZE-1:0];
55  logic wren_b [MEM_BLOCK_SIZE-1:0];
56  logic [WORD_SIZE-1:0] q_b [MEM_BLOCK_SIZE-1:0];
57
58  myRAMInitializer_meminit_22k initializer(
59                  .clock(clk),
60                  .dataout(init_dataout),
61                  .init(init),
62                  .init_busy(init_busy),
63                  .ram_address(init_address),
64                  .ram_wren(init_wren)
65                  );
66
67  genvar row, col;
68  generate
69    for (row = 0; row < MEM_BLOCK_WIDTH; row++) begin : for_row
70      for (col = 0; col < MEM_BLOCK_WIDTH; col++) begin : for_col
71        localparam integer i = row * MEM_BLOCK_WIDTH + col;
72        myRAM2Port RAM_inst(
73                  .clock(clk),
74                  .address_a(address_a[i]),
75                  .data_a(data_a[i]),
76                  .wren_a(wren_a[i]),
77                  .q_a(q_a[i]),
78                  .address_b(address_b[i]),
79                  .data_b(data_b[i]),
80                  .wren_b(wren_b[i]),
81                  .q_b(q_b[i])
82                  );
83        end
84    end
85  endgenerate
86
87  //pipelined read addresses, to mux proper outputs next cycle.
88
89  logic       last_block_wren, last_block_ren, last_block_rcolen;
90  logic [6:0] last_block_row, last_block_col;
91  logic [WORD_SIZE-1:0]last_block_data [USR_BLOCK_SIZE-1:0];
92
93  logic       last_vword_ren;
94  logic [9:0] last_vword_row, last_vword_col;
95
96  always_comb begin
97    vword_data_out=0;
98    for(integer i=0; i<USR_BLOCK_SIZE; i++) block_data_out[i] = 27'b0;
99    for(integer j=0; j<MEM_BLOCK_SIZE; j++) begin
100       wren_a[j]    = 0;
101       wren_b[j]    = 0;
102       address_a[j] = 'x;
103       address_b[j] = 'x;
104       data_a[j]    = 'x;
```

```
105       data_b[j]    = 'x;
106     end
107   if(init_busy) begin
108     for(integer i=0; i<MEM_BLOCK_SIZE; i++) begin
109       address_b[i] = init_address;
110       data_b[i] = init_dataout;
111       wren_b[i] = init_wren;
112     end
113   end else begin : scp
114     logic [6:0] slice_id;
115     logic [ADDR_WIDTH-1:0] addr;
116
117     if((WRITE_DISABLE!=1 && read_vword_en) || (READ_DISABLE!=1 &&
    ↪  write_vword_en)) begin : rwvword
118       for(integer i=0; i<4; i++) begin
119         FatBufferDecoder(vword_row, vword_col+i, slice_id, addr);
120         address_a[slice_id] = addr;
121
122         if(write_vword_en) begin
123           wren_a[slice_id] = 1;
124           data_a[slice_id] = vword_data_in[i*32 +:27];
125         end
126       end
127     end else if((WRITE_DISABLE !=1 && write_block_en) ||
    ↪  (READ_BLOCK_DISABLE!=1 && read_block_en)) begin : rwblock
128       for(integer row_i=0; row_i<USR_BLOCK_WIDTH; row_i++) begin
129         for(integer col_i=0; col_i<USR_BLOCK_WIDTH; col_i++) begin
130           FatBufferDecoder(row_i + block_row * USR_BLOCK_SPACING,
    ↪  col_i + block_col * USR_BLOCK_SPACING, slice_id, addr);
131           address_a[slice_id] = addr;
132         end
133       end
134     end else if(READ_COL_DISABLE !=1 && read_col_as_block_en) begin :
    ↪  rcolen
135       for(integer row_i=0; row_i<USR_BLOCK_SIZE; row_i++) begin
136         FatBufferDecoder(block_row * USR_BLOCK_SIZE + row_i,
    ↪  block_col, slice_id, addr);
137         address_a[slice_id] = addr;
138       end
139     end
140
141     if(WRITE_DISABLE !=1 && last_vword_ren) begin : wvword
142       for(integer i=0; i<4; i++) begin
143         FatBufferDecoder(last_vword_row, last_vword_col+i, slice_id,
    ↪  addr);
144         vword_data_out[i*32 +:27] = q_a[slice_id];
145       end
146     end
147
148     if( (WRITE_DISABLE !=1 && last_block_wren) || (READ_BLOCK_DISABLE
    ↪  != 1 && last_block_ren)) begin : rwlast
149       for(integer row_i=0; row_i<USR_BLOCK_WIDTH; row_i++) begin
150         for(integer col_i=0; col_i<USR_BLOCK_WIDTH; col_i++) begin
```

```
151           FatBufferDecoder(row_i + last_block_row *
    ↪  USR_BLOCK_SPACING, col_i + last_block_col * USR_BLOCK_SPACING,
    ↪  slice_id, addr);
152
153           if(WRITE_DISABLE !=1 && last_block_wren) begin
154             wren_b[slice_id] = 1;
155             data_b[slice_id] = q_a[slice_id] + last_block_data[row_i
    ↪  * USR_BLOCK_WIDTH + col_i];
156             address_b[slice_id] = addr;
157           end else begin //last_block_ren
158             block_data_out[row_i * USR_BLOCK_WIDTH + col_i] =
    ↪  q_a[slice_id];
159           end
160         end
161       end
162     end
163     if(READ_COL_DISABLE!=1 && last_block_rcolen) begin : rcolen_last
164       for(integer row_i=0; row_i<USR_BLOCK_SIZE; row_i++) begin
165         FatBufferDecoder(block_row * USR_BLOCK_SIZE + row_i,
    ↪  block_col, slice_id, addr);
166         block_data_out[row_i] = 0;
167         block_data_out[row_i] = q_a[slice_id];
168       end
169     end
170   end
171 end
172
173 always @(posedge clk) begin
174   last_block_wren <= write_block_en;
175   last_block_ren <= read_block_en;
176   last_block_rcolen <= read_col_as_block_en;
177   last_block_row <= block_row;
178   last_block_col <= block_col;
179   last_block_data <= block_data_in;
180
181   last_vword_ren <= read_vword_en;
182   last_vword_row <= vword_row;
183   last_vword_col <= vword_col;
184 end
185
186 endmodule
```

FatBufferDecoder.sv

```
function void FatBufferDecoder;

  parameter ADDR_WIDTH = 8;
  parameter MEM_BLOCKS_PER_ROW = 32;
  parameter MEM_BLOCK_WIDTH = 8;

  input [6:0] row;
  input [6:0] col;
  output [6:0] slice_id;
  output [ADDR_WIDTH-1:0] address;


  logic [6:0] block_y = row >> $clog2(MEM_BLOCK_WIDTH); // block_y = row_of_bank / MEM_BLOCK_WIDTH
  logic [6:0] block_x = col >> $clog2(MEM_BLOCK_WIDTH);  //block_x = col / MEM_BLOCK_WIDTH;
  logic [6:0] slice_y = row[$clog2(MEM_BLOCK_WIDTH)-1:0]; // slice_y = row_of_bank % MEM_BLOCK_WIDTH
  logic [6:0] slice_x = col[$clog2(MEM_BLOCK_WIDTH)-1:0];  // slice_x = col % MEM_BLOCK_WIDTH

  logic [11:0] block_id = block_y * MEM_BLOCKS_PER_ROW + block_x;
  slice_id = slice_y * MEM_BLOCK_WIDTH + slice_x + block_id; //notice the block_id is added to prevent the slices from lining up with columns

  address = block_id;
endfunction
```

interfaces.sv

```
/*********** Mem control interfaces ******************/
interface AVL;
  logic local_init_done;
  logic avl_readdatavalid;
  logic avl_burstbegin;
  logic avl_wait_request_n;
  logic [25:0] avl_address;
  logic [127:0] avl_readdata;
  logic [127:0] avl_writedata;
  logic avl_write;
  logic avl_read;
  modport Master (input local_init_done,
            avl_readdatavalid,
            avl_wait_request_n,
            avl_readdata,
         output avl_burstbegin,
            avl_address,
            avl_writedata,
            avl_write,
            avl_read);
endinterface

interface READ_BUFFER #(parameter BLOCK_SIZE = 36, parameter
 ↪  WORD_SIZE = 27);
  logic reset;
  logic ready;
  logic col_mode;
  logic [7:0] block_row;
  logic [7:0] block_col;
  logic signed [WORD_SIZE-1:0] block[BLOCK_SIZE-1:0];
  logic load_ddr;
  logic pad;
  logic [25:0] start_address;
  logic [7:0] stride;
  logic [7:0] rows;
  modport TOP (output load_ddr,
                start_address,
                stride,
                rows,
                pad,
                reset,
                col_mode,
                block_row,
                block_col,
           input ready);
  modport ALU (input block);
  modport BUFFER (input load_ddr,
                start_address,
                stride,
                rows,
```

```
                pad,
                reset,
                col_mode,
                block_row,
                block_col,
             output ready,
                block);
endinterface

interface MASK_BUFFER #(parameter WORD_SIZE = 27);
  logic reset;
  logic ready;
  logic signed [WORD_SIZE-1:0] mask[8:0];
  logic load_ddr;
  logic [25:0] start_address;
  modport TOP (output load_ddr,
               start_address,
               reset,
            input ready);
  modport ALU (input mask);
  modport BUFFER (input reset,
                load_ddr,
                start_address,
              output ready,
                mask);
endinterface

interface WRITE_BACK #(parameter BLOCK_SIZE = 16, parameter WORD_SIZE
 ↪  = 27);
  logic reset;
  logic ready;
  logic accumulate;
  logic [7:0] block_row;
  logic [7:0] block_col;
  logic signed [WORD_SIZE-1:0] block [BLOCK_SIZE-1:0];
  logic store_ddr;
  logic [25:0] start_address;
  logic [7:0] stride;
  logic [7:0] rows;
  modport TOP (output store_ddr,
                start_address,
                stride,
                rows,
                reset,
                block_row,
                block_col,
            input ready);
  modport ALU (output block, accumulate);
  modport BUFFER (input reset,
                accumulate,
                block_row,
                block_col,
                block,
                store_ddr,
```

```
103                 start_address,
104                 stride,
105                 rows,
106              output ready);
107 endinterface
108
109 /***************** ALU interface ******************/
110 interface ALU_i;
111   logic execute;
112   logic [2:0] operation;
113   logic rev_mask;
114   logic [1:0] sub_block;
115   logic [15:0] sub_index;
116   logic ready;
117   modport TOP ( output execute,
118                 operation,
119                 rev_mask,
120                 sub_block,
121                 sub_index,
122             input ready);
123   modport ALU ( input execute,
124                 operation,
125                 rev_mask,
126                 sub_block,
127                 sub_index,
128             output ready);
129 endinterface
```

mask_buffer.sv

```
module mask_buffer (
  input iCLK,
  MASK_BUFFER.BUFFER bif, // Interface for top level and ALU
  AVL.Master avl        // DDR3 interface
);

assign avl.avl_burstbegin = avl.avl_write || avl.avl_read;
assign avl.avl_writedata = 'x;
logic [3:0] state;
logic [4:0] write_count;
logic [7:0] index;
logic i;
// 0: idle
// 1: load mask from ddr
always@(posedge iCLK)
begin
  if (bif.reset) begin
    write_count <= 5'b0;
    state <= 0;
    bif.ready <= 1;
    index <= 7'b0;
    avl.avl_write <= 0;
    avl.avl_read <= 0;
    avl.avl_address <= {26{1'b0}};
  end
  else begin
  case (state)
   0 : begin
      if (avl.local_init_done && bif.load_ddr) begin
        avl.avl_address <= bif.start_address;
        state <= 1;
        bif.ready <= 0;
      end
    end
    //Reading from DDR3 to output
   1 : begin
      avl.avl_read <= 1;

      if (!write_count[3])
        write_count <= write_count + 1'b1;

      if (avl.avl_wait_request_n) //Ready to read
        state <= 2;
    end
   2 : begin
      avl.avl_read <= 0;

      if (!write_count[3])
        write_count <= write_count + 1'b1;

```

```
51        if (avl.avl_readdatavalid)
52        begin
53          for(int i=0; i<4; i++) bif.mask[index*4+i] <=
    ↪  avl.avl_readdata[i*32 +:27];
54          state <= 3;
55        end
56      end
57    3 : begin
58        if (write_count[3])
59        begin
60          write_count <= 5'b0;
61
62          //Done reading
63          if (index + 1 > 8) state <= 4;
64          else
65
66          // Read next
67          begin
68            avl.avl_address <= avl.avl_address + 1'b1;
69            state <= 1;
70            index <= index + 1'b1;
71          end
72        end
73      end
74    4 : begin
75        bif.ready <= 1'b1;
76        state <= 0;
77    end
78    default : state <= 0;
79    endcase
80    end
81  end
82
83  endmodule
```

myAddrInMux.qip

```
set_global_assignment -name IP_TOOL_NAME "LPM_MUX"
set_global_assignment -name IP_TOOL_VERSION "15.1"
set_global_assignment -name IP_GENERATED_DEVICE_FAMILY "{Cyclone V}"
set_global_assignment -name VERILOG_FILE [file join
 ↪  $::quartus(qip_path) "myAddrInMux.v"]
set_global_assignment -name MISC_FILE [file join $::quartus(qip_path)
 ↪  "myAddrInMux_bb.v"]
```

myDataOutMux.qip

```
set_global_assignment -name IP_TOOL_NAME "LPM_MUX"
set_global_assignment -name IP_TOOL_VERSION "15.1"
set_global_assignment -name IP_GENERATED_DEVICE_FAMILY "{Cyclone V}"
set_global_assignment -name VERILOG_FILE [file join
 ↪  $::quartus(qip_path) "myDataOutMux.v"]
```

mylpmmult.qip

```
set_global_assignment -name IP_TOOL_NAME "LPM_MULT"
set_global_assignment -name IP_TOOL_VERSION "15.1"
set_global_assignment -name IP_GENERATED_DEVICE_FAMILY "{Cyclone V}"
set_global_assignment -name VERILOG_FILE [file join
 ↪  $::quartus(qip_path) "mylpm_mult.v"]
set_global_assignment -name MISC_FILE [file join $::quartus(qip_path)
 ↪  "mylpm_mult_bb.v"]
```

myRAM2Port.qip

```
set_global_assignment -name IP_TOOL_NAME "RAM: 1-PORT"
set_global_assignment -name IP_TOOL_VERSION "15.1"
set_global_assignment -name IP_GENERATED_DEVICE_FAMILY "{Cyclone V}"
set_global_assignment -name VERILOG_FILE [file join
 ↪  $::quartus(qip_path) "myRAM.v"]
set_global_assignment -name MISC_FILE [file join $::quartus(qip_path)
 ↪  "myRAM_bb.v"]
```

myRAMInitializer.qip

```
set_global_assignment -name IP_TOOL_NAME "RAM initializer"
set_global_assignment -name IP_TOOL_VERSION "15.1"
set_global_assignment -name IP_GENERATED_DEVICE_FAMILY "{Cyclone V}"
set_global_assignment -name VERILOG_FILE [file join
 ↪ $::quartus(qip_path) "myRAMInitializer.v"]
set_global_assignment -name MISC_FILE [file join $::quartus(qip_path)
 ↪ "myRAMInitializer.bsf"]
set_global_assignment -name MISC_FILE [file join $::quartus(qip_path)
 ↪ "myRAMInitializer_inst.v"]
set_global_assignment -name MISC_FILE [file join $::quartus(qip_path)
 ↪ "myRAMInitializer_bb.v"]
set_global_assignment -name MISC_FILE [file join $::quartus(qip_path)
 ↪ "myRAMInitializer.inc"]
set_global_assignment -name MISC_FILE [file join $::quartus(qip_path)
 ↪ "myRAMInitializer.cmp"]
```

read_buffer.sv

```
module read_buffer (
  input iCLK,
  READ_BUFFER.BUFFER bif, // Interface for Top level and ALU
  AVL.Master avl        // DDR3 interface
);

parameter BUFFER_WIDTH = 256;
parameter BLOCK_WIDTH = 6;
parameter BLOCK_SIZE = BLOCK_WIDTH * BLOCK_WIDTH;
parameter WORD_SIZE = 27;

parameter COL_DISABLE = 0;
parameter BLOCK_DISABLE = 0;

assign avl.avl_burstbegin = avl.avl_write || avl.avl_read;
assign avl.avl_writedata = 'x;

logic [3:0] state;
logic [15:0] row;
logic [15:0] index;
logic [4:0] write_count;

logic fb_init_busy;
logic [9:0] vword_row, vword_col;
wire [127:0] vword_data;

wire signed [WORD_SIZE-1:0] fb_block [BLOCK_SIZE-1:0];

FatBuffer #(
        .WORD_SIZE(WORD_SIZE),
        .BUFFER_WIDTH(BUFFER_WIDTH),
        .USR_BLOCK_WIDTH(BLOCK_WIDTH),
        .USR_BLOCK_SPACING(BLOCK_WIDTH-2),
        .WRITE_DISABLE(1),
        .READ_COL_DISABLE(COL_DISABLE),
        .READ_BLOCK_DISABLE(BLOCK_DISABLE)
        )
      fb(.clk(iCLK),
        .init(bif.reset),
        .init_busy(fb_init_busy),
        .read_block_en(~bif.col_mode),
        .read_col_as_block_en(bif.col_mode),
        .write_block_en(0),
        .block_row(bif.block_row),
        .block_col(bif.block_col),
        .block_data_out(fb_block),
        .write_vword_en(avl.avl_readdatavalid),
        .read_vword_en(0),
        .vword_row(vword_row),
        .vword_col(vword_col),
```

```
51          .vword_data_in(vword_data)
52          );
53  assign bif.block = fb_block;
54
55  always_comb begin
56    vword_data = avl.avl_readdata;
57    if(bif.pad) begin
58      vword_row=row + 1'b1;
59      vword_col=index*4 + 1'b1;
60    end else begin
61      vword_row=row;
62      vword_col=index*4;
63    end
64  end
65
66  // 0: idle
67  // 1: load from ddr
68  always@(posedge iCLK)
69  begin
70    if (bif.reset) begin
71      write_count <= 0;
72      index <= 16'b0;
73      row <= 16'b0;
74      state <= 5;
75      bif.ready <= 0;
76      avl.avl_read <= 0;
77      avl.avl_write <= 0;
78    end
79    else begin
80    case (state)
81     0 : begin
82      if (avl.local_init_done && bif.load_ddr) begin
83          avl.avl_address <= bif.start_address;
84          state <= 1;
85          bif.ready <= 0;
86      end
87     end
88    //Reading from DDR3 to local registers
89     1 : begin
90        avl.avl_read <= 1;
91
92        if (!write_count[3])
93          write_count <= write_count + 1'b1;
94
95        if (avl.avl_wait_request_n) //ready to read
96          state <= 2;
97      end
98     2 : begin
99        avl.avl_read <= 0;
100
101       if (!write_count[3])
102         write_count <= write_count + 1'b1;
103
104       if (avl.avl_readdatavalid)
```

```
105         state <= 3;
106       end
107     3 : begin
108         write_count <= 5'b0;
109 
110         //Done reading
111         if ((index >= bif.stride - 1) && (row >= bif.rows - 1)) state
    ↪  <= 4;
112 
113         else // Read next
114         begin
115           avl.avl_address <= avl.avl_address + 1'b1;
116           state <= 1;
117           if (index >= bif.stride - 1)
118           begin
119             index <= 0;
120             row <= row + 1'b1;
121           end else index <= index + 1'b1;
122         end
123       end
124     4 : begin
125       bif.ready <= 1'b1;
126       state <= 0;
127       end
128     //wait for buffer initialization
129     5 : begin
130         if(!fb_init_busy) begin
131           bif.ready <= 1;
132           state <= 0;
133         end
134       end
135     default : state <= 0;
136     endcase
137     end
138 end
139 
140 endmodule
```

relu.sv

```
1  module relu(pixels_in, pixels_out);
2
3  parameter SIZE = 16;
4
5  parameter WORD_SIZE = 27;
6
7  input signed [WORD_SIZE-1:0] pixels_in [(SIZE - 1):0];
8  output signed [WORD_SIZE-1:0] pixels_out [(SIZE - 1):0];
9
10 genvar i;
11 generate for (i = 0; i < SIZE; i++) begin : for_i
12   assign pixels_out[i] = (pixels_in[i] < $signed(0))? 1'b0 :
   ↪  pixels_in[i];
13 end
14 endgenerate
15
16 endmodule
```

reverse_mask.sv

```
module reverse_mask #(parameter WORD_SIZE = 27) (
        input logic signed [WORD_SIZE-1:0] kernel [8:0],
        output logic signed [WORD_SIZE-1:0] reversed [8:0]);

always_comb begin
    reversed[0] = kernel[8];
    reversed[1] = kernel[7];
    reversed[2] = kernel[6];
    reversed[3] = kernel[5];
    reversed[4] = kernel[4];
    reversed[5] = kernel[3];
    reversed[6] = kernel[2];
    reversed[7] = kernel[1];
    reversed[8] = kernel[0];
end
endmodule
```

SoCKit_DDR3_RTL_Test.sv

```
1  /*
2  TOP LEVEL MODULE
3    - module instantiation
4    - Instruction decoder
5
6  Takes in 64-bit instructions from the CPU
7
8  There are two types of instructions: memory control and ALU
   ↪  instructions
9
10
11 This file was previously the SoCKit's DDR3 test module, which
   ↪  explains the strange
12 name. Over time it became the top level module of the project,
   ↪  because
13 the original SoCKit board test had a built-in test which was
   ↪  repurposed
14 at various stages of the project.
15  */
16
17 `define ENABLE_DDR3
18 //`define ENABLE_HPS
19 //`define ENABLE_HSMC_XCVR
20 module SoCKit_DDR3_RTL_Test(
21   select,
22                ///////////AUD/////////////
23                AUD_ADCDAT,
24                AUD_ADCLRCK,
25                AUD_BCLK,
26                AUD_DACDAT,
27                AUD_DACLRCK,
28                AUD_I2C_SCLK,
29                AUD_I2C_SDAT,
30                AUD_MUTE,
31                AUD_XCK,
32
33 `ifdef ENABLE_DDR3
34                ////////DDR3////////
35                DDR3_A,
36                DDR3_BA,
37                DDR3_CAS_n,
38                DDR3_CKE,
39                DDR3_CK_n,
40                DDR3_CK_p,
41                DDR3_CS_n,
42                DDR3_DM,
43                DDR3_DQ,
44                DDR3_DQS_n,
45                DDR3_DQS_p,
46                DDR3_ODT,
```

```
47                DDR3_RAS_n,
48                DDR3_RESET_n,
49                DDR3_RZQ,
50                DDR3_WE_n,
51  `endif /*ENABLE_DDR3*/
52
53                //////////FAN//////////
54                FAN_CTRL,
55
56  `ifdef ENABLE_HPS
57                //////////HPS//////////
58                HPS_CLOCK_25,
59                HPS_CLOCK_50,
60                HPS_CONV_USB_n,
61                HPS_DDR3_A,
62                HPS_DDR3_BA,
63                HPS_DDR3_CAS_n,
64                HPS_DDR3_CKE,
65                HPS_DDR3_CK_n,
66                HPS_DDR3_CK_p,
67                HPS_DDR3_CS_n,
68                HPS_DDR3_DM,
69                HPS_DDR3_DQ,
70                HPS_DDR3_DQS_n,
71                HPS_DDR3_DQS_p,
72                HPS_DDR3_ODT,
73                HPS_DDR3_RAS_n,
74                HPS_DDR3_RESET_n,
75                HPS_DDR3_RZQ,
76                HPS_DDR3_WE_n,
77                HPS_ENET_GTX_CLK,
78                HPS_ENET_INT_n,
79                HPS_ENET_MDC,
80                HPS_ENET_MDIO,
81                HPS_ENET_RESET_n,
82                HPS_ENET_RX_CLK,
83                HPS_ENET_RX_DATA,
84                HPS_ENET_RX_DV,
85                HPS_ENET_TX_DATA,
86                HPS_ENET_TX_EN,
87                HPS_FLASH_DATA,
88                HPS_FLASH_DCLK,
89                HPS_FLASH_NCSO,
90                HPS_GSENSOR_INT,
91                HPS_I2C_CLK,
92                HPS_I2C_SDA,
93                HPS_KEY,
94                HPS_LCM_D_C,
95                HPS_LCM_RST_N,
96                HPS_LCM_SPIM_CLK,
97                HPS_LCM_SPIM_MISO,
98                HPS_LCM_SPIM_MOSI,
99                HPS_LCM_SPIM_SS,
100               HPS_LED,
```

```
101                 HPS_LTC_GPIO,
102                 HPS_RESET_n,
103                 HPS_SD_CLK,
104                 HPS_SD_CMD,
105                 HPS_SD_DATA,
106                 HPS_SPIM_CLK,
107                 HPS_SPIM_MISO,
108                 HPS_SPIM_MOSI,
109                 HPS_SPIM_SS,
110                 HPS_SW,
111                 HPS_UART_RX,
112                 HPS_UART_TX,
113                 HPS_USB_CLKOUT,
114                 HPS_USB_DATA,
115                 HPS_USB_DIR,
116                 HPS_USB_NXT,
117                 HPS_USB_STP,
118                 HPS_WARM_RST_n,
119 `endif /*ENABLE_HPS*/
120
121                 ////////HSMC////////
122                 HSMC_CLKIN_n,
123                 HSMC_CLKIN_p,
124                 HSMC_CLKOUT_n,
125                 HSMC_CLKOUT_p,
126                 HSMC_CLK_IN0,
127                 HSMC_CLK_OUT0,
128                 HSMC_D,
129
130 `ifdef ENABLE_HSMC_XCVR
131
132                 HSMC_GXB_RX_p,
133                 HSMC_GXB_TX_p,
134                 HSMC_REF_CLK_p,
135 `endif
136                 HSMC_RX_n,
137                 HSMC_RX_p,
138                 HSMC_SCL,
139                 HSMC_SDA,
140                 HSMC_TX_n,
141                 HSMC_TX_p,
142
143                 ////////IRDA////////
144                 IRDA_RXD,
145
146                 ////////KEY////////
147                 KEY,
148
149                 ////////LED////////
150                 LED,
151
152                 ////////OSC////////
153                 OSC_50_B3B,
154                 OSC_50_B4A,
```

```
155                 OSC_50_B5B,
156                 OSC_50_B8A,
157
158                 ////////PCIE////////
159                 PCIE_PERST_n,
160                 PCIE_WAKE_n,
161
162                 ////////RESET////////
163                 RESET_n,
164
165                 ////////SI5338////////
166                 SI5338_SCL,
167                 SI5338_SDA,
168
169                 ////////SW////////
170                 SW,
171
172                 ////////TEMP////////
173                 TEMP_CS_n,
174                 TEMP_DIN,
175                 TEMP_DOUT,
176                 TEMP_SCLK,
177
178                 ////////USB////////
179                 USB_B2_CLK,
180                 USB_B2_DATA,
181                 USB_EMPTY,
182                 USB_FULL,
183                 USB_OE_n,
184                 USB_RD_n,
185                 USB_RESET_n,
186                 USB_SCL,
187                 USB_SDA,
188                 USB_WR_n,
189
190                 ////////VGA////////
191                 VGA_B,
192                 VGA_BLANK_n,
193                 VGA_CLK,
194                 VGA_G,
195                 VGA_HS,
196                 VGA_R,
197                 VGA_SYNC_n,
198                 VGA_VS,
199
200 );
201
202 //=======================================================
203 //  PORT declarations
204 //=======================================================
205
206 ////////// AUD //////////
207 input                                                   AUD_ADCDAT;
208 inout                                                   AUD_ADCLRCK;
```

```
209 inout                                   AUD_BCLK;
210 output                                  AUD_DACDAT;
211 inout                                   AUD_DACLRCK;
212 output                                  AUD_I2C_SCLK;
213 inout                                   AUD_I2C_SDAT;
214 output                                  AUD_MUTE;
215 output                                  AUD_XCK;
216
217 `ifdef ENABLE_DDR3
218 ///////// DDR3 /////////
219 output                      [14:0]      DDR3_A;
220 output                      [2:0]       DDR3_BA;
221 output                                  DDR3_CAS_n;
222 output                                  DDR3_CKE;
223 output                                  DDR3_CK_n;
224 output                                  DDR3_CK_p;
225 output                                  DDR3_CS_n;
226 output                      [3:0]       DDR3_DM;
227 inout                       [31:0]      DDR3_DQ;
228 inout                       [3:0]       DDR3_DQS_n;
229 inout                       [3:0]       DDR3_DQS_p;
230 output                                  DDR3_ODT;
231 output                                  DDR3_RAS_n;
232 output                                  DDR3_RESET_n;
233 input                                   DDR3_RZQ;
234 output                                  DDR3_WE_n;
235 `endif /*ENABLE_DDR3*/
236
237 ///////// FAN /////////
238 output                                  FAN_CTRL;
239
240 `ifdef ENABLE_HPS
241 ///////// HPS /////////
242 input                                   HPS_CLOCK_25;
243 input                                   HPS_CLOCK_50;
244 input                                   HPS_CONV_USB_n;
245 output                      [14:0]      HPS_DDR3_A;
246 output                      [2:0]       HPS_DDR3_BA;
247 output                                  HPS_DDR3_CAS_n;
248 output                                  HPS_DDR3_CKE;
249 output                                  HPS_DDR3_CK_n;
250 output                                  HPS_DDR3_CK_p;
251 output                                  HPS_DDR3_CS_n;
252 output                      [3:0]       HPS_DDR3_DM;
253 inout                       [31:0]      HPS_DDR3_DQ;
254 inout                       [3:0]       HPS_DDR3_DQS_n;
255 inout                       [3:0]       HPS_DDR3_DQS_p;
256 output                                  HPS_DDR3_ODT;
257 output                                  HPS_DDR3_RAS_n;
258 output                                  HPS_DDR3_RESET_n;
259 input                                   HPS_DDR3_RZQ;
260 output                                  HPS_DDR3_WE_n;
261 input                                   HPS_ENET_GTX_CLK;
262 input                                   HPS_ENET_INT_n;
```

```
263 output                                  HPS_ENET_MDC;
264 inout                                   HPS_ENET_MDIO;
265 output                          HPS_ENET_RESET_n;
266 input                                   HPS_ENET_RX_CLK;
267 input                       [3:0]       HPS_ENET_RX_DATA;
268 input                                   HPS_ENET_RX_DV;
269 output                      [3:0]       HPS_ENET_TX_DATA;
270 output                                  HPS_ENET_TX_EN;
271 inout                       [3:0]       HPS_FLASH_DATA;
272 output                                  HPS_FLASH_DCLK;
273 output                                  HPS_FLASH_NCSO;
274 input                                   HPS_GSENSOR_INT;
275 inout                                   HPS_I2C_CLK;
276 inout                                   HPS_I2C_SDA;
277 input                       [3:0]       HPS_KEY;
278 output                                  HPS_LCM_D_C;
279 output                                  HPS_LCM_RST_N;
280 input                                   HPS_LCM_SPIM_CLK;
281 inout                                   HPS_LCM_SPIM_MISO;
282 output                                  HPS_LCM_SPIM_MOSI;
283 output                                  HPS_LCM_SPIM_SS;
284 output                      [3:0]       HPS_LED;
285 inout                                   HPS_LTC_GPIO;
286 input                                   HPS_RESET_n;
287 output                                  HPS_SD_CLK;
288 inout                                   HPS_SD_CMD;
289 inout                       [3:0]       HPS_SD_DATA;
290 output                                  HPS_SPIM_CLK;
291 input                                   HPS_SPIM_MISO;
292 output                                  HPS_SPIM_MOSI;
293 output                                  HPS_SPIM_SS;
294 input                       [3:0]       HPS_SW;
295 input                                   HPS_UART_RX;
296 output                                  HPS_UART_TX;
297 input                                   HPS_USB_CLKOUT;
298 inout                       [7:0]       HPS_USB_DATA;
299 input                                   HPS_USB_DIR;
300 input                                   HPS_USB_NXT;
301 output                                  HPS_USB_STP;
302 input                                   HPS_WARM_RST_n;
303 `endif /*ENABLE_HPS*/
304
305 ///////// HSMC ////////
306 inout                       [2:1]       HSMC_CLKIN_n;
307 inout                       [2:1]       HSMC_CLKIN_p;
308 inout                       [2:1]       HSMC_CLKOUT_n;
309 inout                       [2:1]       HSMC_CLKOUT_p;
310 inout                                   HSMC_CLK_IN0;
311 inout                                   HSMC_CLK_OUT0;
312 inout                       [3:0]       HSMC_D;
313
314 `ifdef ENABLE_HSMC_XCVR
315 input                       [7:0]       HSMC_GXB_RX_p;
316 output                      [7:0]       HSMC_GXB_TX_p;
```

```
317 input                                   HSMC_REF_CLK_p;
318 `endif
319
320 inout                       [16:0]      HSMC_RX_n;
321 inout                       [16:0]      HSMC_RX_p;
322 output                                  HSMC_SCL;
323 inout                                   HSMC_SDA;
324 inout                       [16:0]      HSMC_TX_n;
325 inout                       [16:0]      HSMC_TX_p;
326
327 ///////// IRDA /////////
328 input                                   IRDA_RXD;
329
330 ///////// KEY /////////
331 input                       [3:0]       KEY;
332
333 ///////// LED /////////
334 output                      [3:0]       LED;
335
336 ///////// OSC /////////
337 input                                   OSC_50_B3B;
338 input                                   OSC_50_B4A;
339 input                                   OSC_50_B5B;
340 input                                   OSC_50_B8A;
341
342 ///////// PCIE /////////
343 input                                   PCIE_PERST_n;
344 input                                   PCIE_WAKE_n;
345
346 ///////// RESET /////////
347 input                                   RESET_n;
348
349 ///////// SI5338 /////////
350 inout                                   SI5338_SCL;
351 inout                                   SI5338_SDA;
352
353 ///////// SW /////////
354 input                       [3:0]       SW;
355
356 ///////// TEMP /////////
357 output                                  TEMP_CS_n;
358 output                                  TEMP_DIN;
359 input                                   TEMP_DOUT;
360 output                                  TEMP_SCLK;
361
362 ///////// USB /////////
363 input                                   USB_B2_CLK;
364 inout                       [7:0]       USB_B2_DATA;
365 output                                  USB_EMPTY;
366 output                                  USB_FULL;
367 input                                   USB_OE_n;
368 input                                   USB_RD_n;
369 input                                   USB_RESET_n;
370 inout                                   USB_SCL;
```

```
inout                                               USB_SDA;
input                                               USB_WR_n;

//////// VGA ////////
output                      [7:0]                   VGA_B;
output                                              VGA_BLANK_n;
output                                              VGA_CLK;
output                      [7:0]                   VGA_G;
output                                              VGA_HS;
output                      [7:0]                   VGA_R;
output                                              VGA_SYNC_n;
output                                              VGA_VS;


//=======================================================
//  REG/WIRE declarations
//=======================================================
wire afi_clk; // clock for test controllers
/// test status ..
//DDR3 Verify (A)
wire fpga_ddr3_test_pass/*synthesis keep*/;
wire fpga_ddr3_test_fail/*synthesis keep*/;
wire fpga_ddr3_test_complete/*synthesis keep*/;
wire fpga_ddr3_local_init_done/*synthesis keep*/;
wire fpga_ddr3_local_cal_success/*synthesis keep*/;
wire fpga_ddr3_local_cal_fail/*synthesis keep*/;

assign FAN_CTRL = 1'bz;

//=======================================================
//  Structural coding
//=======================================================
  fpga_ddr3 fpga_ddr3_inst(
    /*input  wire         */    .pll_ref_clk(OSC_50_B4A),
    /*input  wire         */
 ↪  .global_reset_n(test_global_reset_n),
    /*input  wire         */
 ↪  .soft_reset_n(test_software_reset_n),
    /*output wire         */    .afi_clk(),
    /*output wire         */    .afi_half_clk(afi_clk),
    /*output wire         */    .afi_reset_n(),
    /*output wire [14:0]  */    .mem_a(DDR3_A),
    /*output wire [2:0]   */    .mem_ba(DDR3_BA),
    /*output wire [0:0]   */    .mem_ck(DDR3_CK_p),
    /*output wire [0:0]   */    .mem_ck_n(DDR3_CK_n),
    /*output wire [0:0]   */    .mem_cke(DDR3_CKE),
    /*output wire [0:0]   */    .mem_cs_n(DDR3_CS_n),
    /*output wire [3:0]   */    .mem_dm(DDR3_DM),
    /*output wire [0:0]   */    .mem_ras_n(DDR3_RAS_n),
    /*output wire [0:0]   */    .mem_cas_n(DDR3_CAS_n),
    /*output wire [0:0]   */    .mem_we_n(DDR3_WE_n),
    /*output wire         */    .mem_reset_n(DDR3_RESET_n),
    /*inout  wire [31:0]  */    .mem_dq(DDR3_DQ),
    /*inout  wire [3:0]   */    .mem_dqs(DDR3_DQS_p),
```

```
423     /*inout  wire [3:0]   */     .mem_dqs_n(DDR3_DQS_n),
424     /*output wire [0:0]   */     .mem_odt(DDR3_ODT),
425
426     /*output wire         */     .avl_ready_0(fpga_ddr3_avl_ready),
427     /*input  wire         */
    ↪   .avl_burstbegin_0(fpga_ddr3_avl_burstbegin),
428     /*input  wire [25:0]  */     .avl_addr_0(fpga_ddr3_avl_addr),
429     /*output wire         */
    ↪   .avl_rdata_valid_0(fpga_ddr3_avl_rdata_valid),
430     /*output wire [127:0] */     .avl_rdata_0(fpga_ddr3_avl_rdata),
431     /*input  wire [127:0] */     .avl_wdata_0(fpga_ddr3_avl_wdata),
432     /*input  wire [15:0]  */     .avl_be_0(16'hFFFF),
433     /*input  wire         */
    ↪   .avl_read_req_0(fpga_ddr3_avl_read_req),
434     /*input  wire         */
    ↪   .avl_write_req_0(fpga_ddr3_avl_write_req),
435     /*input  wire [2:0]   */     .avl_size_0(fpga_ddr3_avl_size),
436     /*output wire         */
    ↪   .local_init_done(fpga_ddr3_local_init_done),
437     /*output wire         */
    ↪   .local_cal_success(fpga_ddr3_local_cal_success),
438     /*output wire         */
    ↪   .local_cal_fail(fpga_ddr3_local_cal_fail),
439
440     /*input  wire         */     .oct_rzqin(DDR3_RZQ)
441   );
442
443   ////////////////// DDR3(A) Test //////////////////
444 wire         fpga_ddr3_avl_ready;                    //
    ↪  avl.waitrequest_n
445 wire         fpga_ddr3_avl_burstbegin;               //
    ↪  .beginbursttransfer
446 wire [25:0]  fpga_ddr3_avl_addr;                     //
    ↪  .address
447 wire         fpga_ddr3_avl_rdata_valid;              //
    ↪  .readdatavalid
448 wire [127:0] fpga_ddr3_avl_rdata;                    //
    ↪  .readdata
449 wire [127:0] fpga_ddr3_avl_wdata;                    //
    ↪  .writedata
450 wire         fpga_ddr3_avl_read_req;                 //
    ↪  .read
451 wire         fpga_ddr3_avl_write_req;                //
    ↪  .write
452 wire [2:0]   fpga_ddr3_avl_size;                     //
    ↪  .burstcount
453
454 assign fpga_ddr3_avl_size = 3'b001;
455
456 parameter WORD_SIZE = 27;
457
458 ////////////////////////////////////////////
459 // Interface declarations
460 AVL to_ddr3();
```

```
461 READ_BUFFER #(.WORD_SIZE(WORD_SIZE)) read_buff1();
462 READ_BUFFER #(.WORD_SIZE(WORD_SIZE)) read_buff2();
463 MASK_BUFFER #(.WORD_SIZE(WORD_SIZE)) mask_buff();
464 WRITE_BACK #(.WORD_SIZE(WORD_SIZE)) accumulator();
465 ALU_i alu_i();
466
467 ////////////////////////////////////////////
468 // Connect AVL interface
469 always_comb begin
470   to_ddr3.local_init_done    = fpga_ddr3_local_init_done;
471   to_ddr3.avl_readdatavalid  = fpga_ddr3_avl_rdata_valid;
472   to_ddr3.avl_wait_request_n = fpga_ddr3_avl_ready;
473   to_ddr3.avl_readdata       = fpga_ddr3_avl_rdata;
474   fpga_ddr3_avl_burstbegin   = to_ddr3.avl_burstbegin;
475   fpga_ddr3_avl_addr      = to_ddr3.avl_address;
476   fpga_ddr3_avl_wdata      = to_ddr3.avl_writedata;
477   fpga_ddr3_avl_write_req    = to_ddr3.avl_write;
478   fpga_ddr3_avl_read_req    = to_ddr3.avl_read;
479 end
480 ////////////////////////////////////////////
481 // Memory control unit
482 input [3:0] select;
483 mem_control #(.WORD_SIZE(WORD_SIZE))  mv(
484   .iCLK(afi_clk),
485   .selector(select),
486   .to_ddr3(to_ddr3.Master),
487   .rbi1(read_buff1.BUFFER),
488   .rbi2(read_buff2.BUFFER),
489   .mi(mask_buff.BUFFER),
490   .wi(accumulator.BUFFER)
491 );
492
493
494 ////////////////////////////////////////////
495 // ALU unit
496 ALU #(.WORD_SIZE(WORD_SIZE)) alu (
497   .iCLK(afi_clk),
498   .in_block1(read_buff1.ALU),
499   .in_block2(read_buff2.ALU),
500   .mask(mask_buff.ALU),
501   .out_block(accumulator.ALU),
502   .from_top(alu_i.ALU)
503 );
504
505 ////////////////////////////////////////////
506 // Top Level Instructions
507
508 logic[63:0] instruction;
509
510 logic cpu_ready;
511
512 //common instruction decoing
513 logic [3:0] inst_op;
514
```

```
515 assign inst_op = instruction[63:60];
516
517 //Memory instruction decoding
518 logic        inst_reset;
519 logic        inst_pad;
520 logic [25:0] inst_addr;
521 logic  [7:0] zzz;
522 logic  [7:0] inst_rows;
523 logic  [7:0] inst_cols;
524 logic  [7:0] inst_stride;
525
526 assign {inst_reset, inst_pad, inst_addr, zzz, inst_rows, inst_cols,
    ↪  inst_stride} = instruction[57:0];
527
528
529
530 assign read_buff1.pad = inst_pad;
531 assign read_buff2.pad = inst_pad;
532
533 assign read_buff1.start_address  = inst_addr;
534 assign read_buff2.start_address  = inst_addr;
535 assign mask_buff.start_address   = inst_addr;
536 assign accumulator.start_address = inst_addr;
537
538 assign read_buff1.rows  = inst_rows;
539 assign read_buff2.rows  = inst_rows;
540 assign accumulator.rows = inst_rows;
541
542 assign read_buff1.stride  = inst_stride;
543 assign read_buff2.stride  = inst_stride;
544 assign accumulator.stride = inst_stride;
545
546
547 //ALU instruction decoding
548 logic [2:0]  inst_alu_op;
549 logic [15:0] inst_sub_index;
550 logic [1:0]  inst_sub_block;
551 logic        inst_rev_mask;
552 logic        inst_block_in_col_modde;
553 logic [7:0]  inst_block_in_row;
554 logic [7:0]  inst_block_in_col;
555 logic [7:0]  inst_block_out_row;
556 logic [7:0]  inst_block_out_col;
557
558 assign inst_alu_op = instruction[58:56];
559
560 assign inst_block_in_col_mode = instruction[48];
561 assign inst_sub_index = instruction[47:32];
562 assign inst_sub_block = instruction[33:32];
563 assign inst_rev_mask  = instruction[32];
564
565 assign inst_block_in_row  = instruction[31:24];
566 assign inst_block_in_col  = instruction[23:16];
567 assign inst_block_out_row = instruction[15:8];
```

```
568 assign inst_block_out_col = instruction[7:0];
569
570 assign alu_i.operation = inst_alu_op;
571 assign alu_i.sub_index = inst_sub_index;
572 assign alu_i.sub_block = inst_sub_block;
573 assign alu_i.rev_mask  = inst_rev_mask;
574
575 assign read_buff1.col_mode    = inst_block_in_col_mode;
576 assign read_buff2.col_mode    = inst_block_in_col_mode;
577
578 assign read_buff1.block_row  = inst_block_in_row;
579 assign read_buff2.block_row  = inst_block_in_row;
580 assign accumulator.block_row = inst_block_out_row;
581
582 assign read_buff1.block_col  = inst_block_in_col;
583 assign read_buff2.block_col  = inst_block_in_col;
584 assign accumulator.block_col = inst_block_out_col;
585
586 logic[7:0] state;
587
588 always @(posedge afi_clk)
589 begin
590 read_buff1.reset  = 0;
591 read_buff2.reset  = 0;
592 mask_buff.reset   = 0;
593 accumulator.reset = 0;
594
595 read_buff1.load_ddr    = 0;
596 read_buff2.load_ddr    = 0;
597 mask_buff.load_ddr     = 0;
598 accumulator.store_ddr  = 0;
599 alu_i.execute          = 0;
600
601 if(test_software_reset_n) begin
602   state             <= 0;
603   cpu_ready         <= 1;
604
605 end
606 case(state)
607   0: begin
608     case(inst_op)
609       0: begin //noop
610         if(inst_reset) begin
611           read_buff1.reset  = 1;
612           read_buff2.reset  = 1;
613           mask_buff.reset   = 1;
614           accumulator.reset = 1;
615           state <= 6;
616         end
617       end
618       1: begin //read buffer1
619         if(inst_reset)
620           read_buff1.reset  = 1;
621         else
```

```
622             read_buff1.load_ddr = 1;
623           cpu_ready <= 0;
624           state <= 1;
625         end
626         2: begin //read buffer2
627           if(inst_reset)
628             read_buff2.reset  = 1;
629           else
630             read_buff2.load_ddr = 1;
631           cpu_ready <=0;
632           state <= 2;
633         end
634         3: begin //mask buffer
635           mask_buff.load_ddr = 1;
636           cpu_ready <=0;
637           state <= 3;
638         end
639         4: begin //write buffer
640           accumulator.store_ddr = 1;
641           cpu_ready <=0;
642           state <= 4;
643         end
644         5: begin //ALU operation
645           alu_i.execute = 1;
646           cpu_ready <=0;
647           state <= 5;
648         end
649       endcase
650     end
651     1: begin
652       if(read_buff1.ready) begin
653         cpu_ready <= 1;
654         state    <= 0;
655       end else
656         read_buff1.load_ddr    = 1;
657     end
658     2: begin
659       if(read_buff2.ready) begin
660         cpu_ready <= 1;
661         state     <= 0;
662       end else
663         read_buff2.load_ddr    = 1;
664     end
665     3: begin
666
667       if(mask_buff.ready) begin
668         cpu_ready <= 1;
669         state    <= 0;
670       end else
671         mask_buff.load_ddr     = 1;
672     end
673     4: begin
674       if(accumulator.ready) begin
675         cpu_ready <= 1;
```

```
676         state    <= 0;
677       end else
678         accumulator.store_ddr  = 1;
679     end
680     5: begin
681       if(alu_i.ready) begin
682         cpu_ready <= 1;
683         state    <= 0;
684       end else
685         alu_i.execute          = 1;
686     end
687     6: begin
688       if(read_buff1.ready && read_buff2.ready && mask_buff.ready &&
      ↪  accumulator.ready) begin
689         cpu_ready <= 1;
690         state     <= 0;
691       end
692     end
693   endcase
694   end
695
696   always @ (posedge OSC_50_B3B)
697   begin
698     instruction <= instruction + 1;
699   end
700
701
702   endmodule
```

soft_lpmmult.qip

```
set_global_assignment -name IP_TOOL_NAME "LPM_MULT"
set_global_assignment -name IP_TOOL_VERSION "15.1"
set_global_assignment -name IP_GENERATED_DEVICE_FAMILY "{Cyclone V}"
set_global_assignment -name VERILOG_FILE [file join
 ↪  $::quartus(qip_path) "soft_lpmmult.v"]
set_global_assignment -name MISC_FILE [file join $::quartus(qip_path)
 ↪  "soft_lpmmult_bb.v"]
```

vector_multiplexer.sv

```
module vector_multiplexer
  #(parameter WIDTH = 27,
    parameter INPUTS = 2,
    parameter SELECT_WIDTH = $clog2(INPUTS)
  )
  (
      input [WIDTH-1:0] in [INPUTS-1:0],
      input [SELECT_WIDTH-1:0] sel,
      output[WIDTH-1:0] out
  );

  always_comb begin
    out = in[sel];
  end

endmodule
```

write_back_accumulator.sv

```
1  module write_back_accumulator (
2    input iCLK,
3    WRITE_BACK.BUFFER bif,   //Interface for top level and ALU
4    AVL.Master avl       //DDR3 interface
5  );
6
7  parameter BUFFER_WIDTH = 8;
8  parameter BLOCK_WIDTH = 4;
9  parameter BLOCK_SIZE = BLOCK_WIDTH*BLOCK_WIDTH;
10 parameter WORD_SIZE = 27;
11 assign avl.avl_burstbegin = avl.avl_write || avl.avl_read;
12
13 logic fb_init_busy;
14
15 wire signed [WORD_SIZE-1:0] block_data [BLOCK_SIZE-1:0];
16
17 logic read_vword_en;
18
19 wire [127:0] vword_data;
20 logic [9:0] vword_row, vword_col;
21
22 FatBuffer #(
23         .WORD_SIZE(WORD_SIZE),
24         .USR_BLOCK_WIDTH(BLOCK_WIDTH),
25         .USR_BLOCK_SPACING(BLOCK_WIDTH),
26         .READ_COL_DISABLE(1),
27         .READ_BLOCK_DISABLE(1)
28         )
29       fb(.clk(iCLK),
30         .init(bif.reset),
31         .init_busy(fb_init_busy),
32         .read_block_en(0),
33         .read_col_as_block_en(0),
34         .write_block_en(bif.accumulate),
35         .block_row(bif.block_row),
36         .block_col(bif.block_col),
37         .block_data_in(block_data),
38         .write_vword_en(0),
39         .read_vword_en(read_vword_en),
40         .vword_row(vword_row),
41         .vword_col(vword_col),
42         .vword_data_in('x),
43         .vword_data_out(vword_data)
44         );
45
46 reg [3:0] state;
47 reg [15:0] index;
48 reg [15:0] row;
49 reg [4:0] write_count;
50
```

```
always_comb begin
  block_data = bif.block;
  avl.avl_writedata = vword_data;
  vword_row = row;
  vword_col = 4*index;

  if(state == 1) read_vword_en = 1;
  else  read_vword_en = 0;
end
// 0 : idle
// 1 : write to ddr
always@(posedge iCLK)
begin
  if (bif.reset) begin
    //Reset signals
    state <= 5;
    write_count <= 0;
    index <= 0;
    row <= 0;
    bif.ready <= 0;
    avl.avl_write <= 0;
    avl.avl_read <= 0;
    avl.avl_address <= {26{1'b0}};
  end
  begin
  case (state)
   0 : begin
      if (avl.local_init_done && bif.store_ddr) begin
        avl.avl_address <= bif.start_address;
        state <= 1;
        bif.ready <= 0;
      end
    end
  // Writing data back to DDR3
   1: begin
      //there is a one cycle delay fetching the avl_writedata, but
 ↪  we will not send the avl_write signal for many cycles, so it's
 ↪  ok
      if (write_count[3])
      begin
        write_count <= 5'b0;
        avl.avl_write <= 1'b1;
        state <= 2;
      end
      else write_count <= write_count + 1'b1;
    end
   2 : begin
      if (avl.avl_wait_request_n)
      begin
          avl.avl_write <= 1'b0;
          state <= 3;
        end
    end
   3 : begin
```

```
103       if ((index >= bif.stride - 1) && (row >= bif.rows - 1)) //
    ↪  Loaded all memory
104       begin
105         avl.avl_address <= {26{1'b0}};
106         state <= 4;
107       end
108       else
109       begin
110         avl.avl_address <= avl.avl_address + 1'b1;
111         state <= 1;
112         if (index >= bif.stride - 1)
113         begin
114           index <= 0;
115           row <= row + 1'b1;
116         end else index <= index + 1'b1;
117       end
118     end
119    4 : begin
120       bif.ready <= 1'b1;
121       state <= 0;
122      end
123    5 : begin
124       if(fb_init_busy) begin
125         bif.ready <= 1;
126         state <= 0;
127       end
128     end
129    default : state <= 0;
130    endcase
131   end
132 end
133
134 endmodule
```

vga_framebuffer.sv

```
module VGA_FB(input logic         clk,
              input logic          reset,
              input logic          write,
              input                chipselect,
              input logic [31:0] address,
         input logic [31:0] writedata,

              output logic [7:0] VGA_R, VGA_G, VGA_B,
              output logic       VGA_CLK, VGA_HS, VGA_VS,
 ↪  VGA_BLANK_n,
              output logic       VGA_SYNC_n);

        logic pixel_write;
        logic[9:0] x,y;
  logic[23:0] rgb;

   VGA_FB_Emulator fb_emulator(.clk50(clk), .*);
  typedef enum logic[2:0] {RESET, HOLD, RUN} state_t;
  state_t state;

  logic [307199:0][23:0] framebuffer;

  logic[18:0]  write_address, read_address;

  assign write_address = address[18:0];
        assign read_address = x + (y << 9) + (y << 7 );
  assign rgb = framebuffer[read_adress];

  logic[18:0] i;
   always_ff @(posedge clk) begin
    if (reset) state <= RESET;
    else case (state)
      RESET: begin
        for (i=18'd0; i < 18'd307199; i = i+ 18'd1) begin
          framebuffer[i] <= 23'd0;
        end
        pixel_write <= 0;
        state <= HOLD;
        end
      HOLD:
        if (chipselect && write) begin
          pixel_write <= 0;
          framebuffer[write_address] <= writedata[24:1];
          if (writedata[1]) state <= RUN;
        end
      RUN:
        pixel_write <= 1;
        state <= HOLD;

      default:
```

```
50          state <= HOLD;
51        endcase
52    end
53  endmodule
```

vga_fb_emulator.sv

```
module VGA_FB_Emulator(
 input logic       clk50, reset,
 input logic [23:0] rgb,
 input logic      pixel_write,
 output logic [9:0] x,y,
 output logic [7:0] VGA_R, VGA_G, VGA_B,
 output logic       VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
 ↪  VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other
 ↪  cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *             _______________              ________
 * ___________|    Video      |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *       _______________________      _____________
 * |____|       VGA_HS          |____|
 */
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
 ↪  HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
 ↪  VBACK_PORCH; // 525

   logic [10:0]           hcount; // Horizontal counter
                                                // Hcount[10:1]
 ↪  indicates pixel column (0-639)
   logic            endOfLine;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else                hcount <= hcount + 11'd 1;

   assign endOfLine = hcount == HTOTAL - 1;

```

```
46    // Vertical counter
47    logic [9:0]             vcount;
48    logic             endOfField;
49
50    always_ff @(posedge clk50 or posedge reset)
51      if (reset)          vcount <= 0;
52      else if (endOfLine)
53        if (endOfField)   vcount <= 0;
54        else              vcount <= vcount + 10'd 1;
55
56    assign endOfField = vcount == VTOTAL - 1;
57
58    // Horizontal sync: from 0x520 to 0x5DF (0x57F)
59    // 101 0010 0000 to 101 1101 1111
60    assign VGA_HS = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] ==
      3'b111));
61    assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);
62
63    assign VGA_SYNC_n = 1; // For adding sync to video signals; not
      used for VGA
64
65    // Horizontal active: 0 to 1279     Vertical active: 0 to 479
66    // 101 0000 0000  1280          01 1110 0000  480
67    // 110 0011 1111  1599          10 0000 1100  524
68    assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
69       !( vcount[9] | (vcount[8:5] == 4'b1111) );
70
71    /* VGA_CLK is 25 MHz
72     *             __    __    __
73     * clk50    __|  |__|  |__|
74     *
75     *             _____       __
76     * hcount[0]__|     |_____|
77     */
78    assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on
      rising edge
79
80    assign VGA_R= pixel_write ? rgb[7:0] : 8'd0;
81    assign VGA_G= pixel_write ? rgb[15:8] : 8'd0;
82    assign VGA_B= pixel_write ? rgb[23:16] : 8'd0;
83
84    always_comb begin
85   if (hcount>11'd1279)
86     x=10'd0;
87   else
88     x=hcount[10:1];
89   if (vcount>10'd479)
90     y=10'd0;
91   else
92     y=vcount;
93    end
94 endmodule
```