Sobel Operator FPGA Design

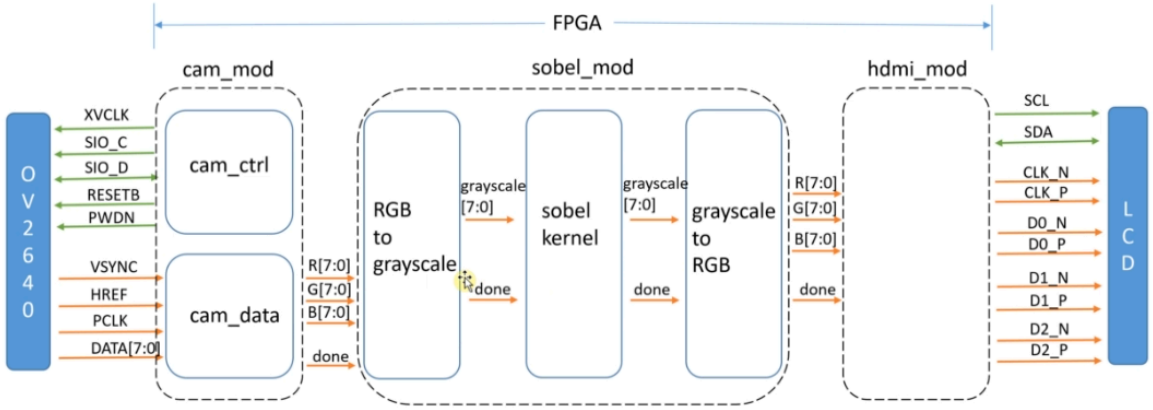
Author	Date
Shuran Xu	2025/03/22

1.Design Overview....1
2. RGB to Grayscale....2
2.1 Technique #1: Scale and down-scale....2
2.2 Technique #2: Express floating-point numbers in terms of shifts....3
3. Reading/writing the BMP file....5
4. RGB-To-Grayscale Simulation....9
5. Sobel Operator Analysis....10
6. Sobel Kernel Interface Analysis....12
7. FIFO Line Buffer Implementation....15
8. Sobel Data Modulation Design....18
9. Sobel Data Buffer Implementation....22
10. Sobel Operator Calculation Implementation....24
11. Grayscale to RGB Implementation....27
12. Sobel Module Wrapper Implementation....28
13. Sobel Module Simulation....30

1.Design Overview

This project designs a sobel edge detector that processes an RGB image and outputs the processed RGB image in black and white where edges are enhanced. The module accepts images in .bmp format and produces the processed image in .bmp format as well. Although this project is only targeting the sobel module itself, it can be extended to include peripheral components such as camera and HDMI modules as the input and output modules for the sobel module, respectively.

The following diagram shows the overall system diagram for this project where we focus on the ‘sobel_mod’ part:



As can be seen above, the sobe_mod module consists of three submodules:

- RGB-to-grayscale: convert the input RGB pixels to grayscale pixels
- Sobel-kernel: performs the sobel kernel filtering on grayscale pixels
- grayscale-to-RGB: convert the processed grayscale pixels to RGB pixels

Since the sobel_mod module accepts images from the camera inputs in a real world scenario, the sobel_mod is designed as a streaming or data-flow circuit where new input images are coming to the sobel_mod module while the module produces the processed images.

2. RGB to Grayscale

To convert an RGB image to a grayscale image, we need to perform the following equations:

Grayscale = 0.299 * R + 0.587 * G + 0.114 * B

Since it is expensive to do floating point computation on FPGA we need to use some techniques to approximate such a computation.

2.1 Technique #1: Scale and down-scale

The idea is to convert all floating-point coefficients into integers, then do the required calculations, and finally shift down the computed value.

In this case, we can first round coefficients into its nearest floating point numbers up to 2 decimal points:

Y = 0.3 * R + 0.59 * G + 0.11 * B

Then we multiply each coefficient by 2^10, which is 1024. We choose power of 2 to ease the downscale step, and we choose 1024 specifically to ensure the fractional points in the scaled number is negligible in terms of accuracy.

1024 * 0.3 = 307.2
1024 * 0.59 = 604.16
1024 * 0.11 = 112.64

Here we can simply ignore the fractional points and get:

Y = 307 * R + 604 * G + 113 * B

Y is the scaled version so we need to shift it down to get the grayscale value:

Y = (307 * R + 604 * G + 113 * B) >> 10;

2.2 Technique #2: Express floating-point numbers in terms of shifts

The technique #1 loses accuracy to ease the computation, an alternative is to express the floating-point coefficients in a much more accurate way by expressing a number in terms of the combination of shifting operations.

Grayscale = 0.299 * R + 0.587 * G + 0.114 * B

0.299 can be approximated to 0.28125:

$0.28125 = 1/2^2 + 1/2^5$

Similarly, we have:

$0.587 \cong 0.5 + 0.0625 = 1/2 + 1/2^4$

$0.114 \cong 0.0625 + 0.03125 = 1/2^4 + 1/2^5$

So now we have:

Y = R * (0.25 + 0.03125) + G * (0.5 + 0.0625) + B * (0.0625 + 0.03125)
$= R * (1/2^2 + 1/2^5) + G * (1/2 + 1/2^4) + B * (1/2^4 + 1/2^5)$

Since each coefficient component now is a power of 2, we can achieve them by right-shifting operation, so the Y expression now becomes:

Y = (R >> 2) + (R >> 5) + (G >> 1) + (G >> 4) + (B >> 4) + (B >> 5)

This is the preferred way to compute grayscale values from RGB values.

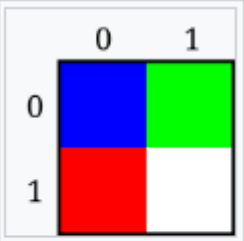
We can implement such an equation in Verilog to construct the rgb_to_grayscale module. Apart from data conversion, we need to use the input done_i signal to control the module so that we can enable this module when needed.

```
//rgb_to_grayscale.v

`timescale 1ps/1ps

module rgb_to_grayscale(

    input clk,
    input rst,

    input [7:0] red_i,
    input [7:0] green_i,
    input [7:0] blue_i,

    input cam_done_i,

    output reg [7:0] grayscale_o,
    output reg done_o
);

reg [15:0] count;

    always @(posedge clk) begin
        if(rst) begin
            count <= 12'b0;
            done_o <= 1'b0;
            grayscale_o <= 7'b0;
        end else begin
            count <= count + 1;
            if(cam_done_i == 1'b1) begin
                grayscale_o <= (red_i >> 2) + (red_i >> 5) +
                               (green_i >> 1) + (green_i >> 4) +
                               (blue_i >> 4) + (blue_i >> 5);
                done_o <= 1'b1;
            end else begin
                grayscale_o <= 8'b0;
                done_o <= 1'b0;
                count <= 12'b0;
```

```
        end
      end
   end

endmodule
```

3. Reading/writing the BMP file

To be able to read BMP files, we need to know the BMP file format so that we can parse the file header fields and file data. Following is an example of a 2×2 pixel, 24-bit bitmap (Windows DIB header BITMAPINFOHEADER) with pixel format RGB24, from Wikipedia:



Example 1 of a 2×2 pixel bitmap, with 24 bits/pixel encoding

Offset	Size	Hex value	Value	Description
BMP Header				
0h	2	42 4D	"BM"	ID field (42h, 4Dh)
2h	4	46 00 00 00	70 bytes (54+16)	Size of the BMP file (54 bytes header + 16 bytes data)
6h	2	00 00	Unused	Application specific
8h	2	00 00	Unused	Application specific
Ah	4	36 00 00 00	54 bytes (14+40)	Offset where the pixel array (bitmap data) can be found
DIB Header				
Eh	4	28 00 00 00	40 bytes	Number of bytes in the DIB header (from this point)
12h	4	02 00 00 00	2 pixels (left to right order)	Width of the bitmap in pixels
16h	4	02 00 00 00	2 pixels (bottom to top order)	Height of the bitmap in pixels. Positive for bottom to top pixel order.
1Ah	2	01 00	1 plane	Number of color planes being used
1Ch	2	18 00	24 bits	Number of bits per pixel
1Eh	4	00 00 00 00	0	BI_RGB, no pixel array compression used
22h	4	10 00 00 00	16 bytes	Size of the raw bitmap data (including padding)
26h	4	13 0B 00 00	2835 pixels/metre horizontal	Print resolution of the image, 72 DPI × 39.3701 inches per metre yields 2834.6472
2Ah	4	13 0B 00 00	2835 pixels/metre vertical	
2Eh	4	00 00 00 00	0 colors	Number of colors in the palette
32h	4	00 00 00 00	0 important colors	0 means all colors are important
Start of pixel array (bitmap data)				
36h	3	00 00 FF	0 0 255	Red, Pixel (x=0, y=1)
39h	3	FF FF FF	255 255 255	White, Pixel (x=1, y=1)
3Ch	2	00 00	0 0	Padding for 4 byte alignment (could be a value other than zero)
3Eh	3	FF 00 00	255 0 0	Blue, Pixel (x=0, y=0)
41h	3	00 FF 00	0 255 0	Green, Pixel (x=1, y=0)
44h	2	00 00	0 0	Padding for 4 byte alignment (could be a value other than zero)

https://en.wikipedia.org/wiki/BMP_file_format

We then implement a task readBMP in SystemVerilog to read a .bmp file and parse various fields as well as read the image data to a local array ‘bmp_data’. Given the input image with dimension 128x128 with 24 bits per pixel, we need at least:

128x128x24 / 8 / 1024 = 48

So the minimum size for bmp_data is 48KB, we can round it up to 50 KB.
To read the file size we see:

2h	4	46 00 00 00	70 bytes (54+16)

This means we need to extract 4 bytes starting from the offset 2, and since the MSB is 0x46, we can concatenate these 4 bytes and assign them to an integer variable such as ‘bmp_size’:

```
bmp_size = {bmp_data[5], bmp_data[4], bmp_data[3], bmp_data[2]};
$display("bmp_size = %d\n", bmp_size);
```

Similarly, we can read start position, width of bitmap in pixels, etc. Here is the Systemverilog code of readBMP task:

```
task readBMP;
    integer fileId;
    integer i;
    begin
        fileId = $fopen(`read_file_name, "rb");
        if(fileId == 0) begin
            $display("Open BMP error\n");
            $finish;
        end else begin
            read_count = $fread(bmp_data, fileId);
            if(read_count < 0) begin
                $display("Failed to read %s\n", `read_file_name);
                $finish;
            end
            $fclose(fileId);

            // extract fields from bmp_data
            bmp_size = {bmp_data[5], bmp_data[4], bmp_data[3], bmp_data[2]};
            $display("bmp_size = %d\n", bmp_size);

            bmp_start_pos = {bmp_data[13], bmp_data[12], bmp_data[11], bmp_data[10]};
            $display("bmp_start_pos = %d\n", bmp_start_pos);

            bmp_width = {bmp_data[21], bmp_data[20], bmp_data[19], bmp_data[18]};
            $display("bmp_width = %d\n", bmp_width);
```

```
        bmp_height = {bmp_data[25], bmp_data[24], bmp_data[23], bmp_data[22]};
        $display("bmp_height = %d\n", bmp_height);

        bit_count = {bmp_data[29], bmp_data[28]};
        $display("bit_count = %d\n", bit_count);

        if(bit_count != 24) begin
           $display("bit_count needs to be 24 !\n");
           $finish;
        end

        // this is the case where we need to pad zeros
        if(bmp_width % 4 == 1) begin
           $display("bmp_width mod 4 needs to be zero\n");
           $finish;
        end

        // for(i = bmp_start_pos; i <bmp_size; i = i + 1) begin
        //     $display("%h", bmp_data[i]);
        // end
      end
   end
endtask
```

Similar to reading a BMP image, we can also write a BMP image by writing the required fields and data. The implementation body is basically a for loop that writes each byte in bmp_data to a file handler:

```
task writeBMPFromGrayscale();
   integer fileId, i;
   begin
      fileId = $fopen(`write_file_name, "wb");
      if(fileId == 0) begin
         $display("Open BMP error\n");
         $finish;
      end

      // write the BMP file header
      for(i = 0; i < bmp_start_pos; i = i + 1) begin
         $fwrite(fileId, "%c", bmp_data[i]);
      end

      // write the BMP file content
      for(i = bmp_start_pos; i < bmp_size; i = i + 1) begin
         $fwrite(fileId, "%c", result[i - bmp_start_pos]);
      end
```

```
        $fclose(fileId);
        $display("writeBMPFromGrayscale completed\n");
    end
endtask
```

4. RGB-To-Grayscale Simulation

So far we have implemented readBMP, writeBMP tasks and the rgb_to_grayscale module, we can use these three items to construct a unit test to exercise the rgb_to_grayscale module.

The simulation flow is straightforward:

- Step 1: read an BMP image
- Step 2: enable rgb_to_grayscale module to perform image format conversion
- Step 3: write the converted image data to an new BMP file

The enabling of the rgb_to_grayscale module can be done by controlling the input signal `done_i`. Recall the interface of the rgb_to_grayscale module is defined as:

```
module rgb_to_grayscale(

    input clk,
    input rst,

    input [7:0] red_i,
    input [7:0] green_i,
    input [7:0] blue_i,

    input cam_done_i,

    output reg [7:0] grayscale_o,
    output reg done_o
);
```

The following is the code snippet that implements the entire simulation flow:

```
// inside rgb_to_grayscale_tb.sv

integer i;
initial begin
    rst = 1'b1;
    done_i = 1'b0;
```

```
    red_i = 8'b0;
    green_i = 8'b0;
    blue_i = 8'b0;

    $dumpfile("waveforms/rgb_to_grayscale.vcd");
    $dumpvars(0, rgb_to_grayscale_tb);

    readBMP;

    #(`clk_period);
    rst = 1'b0;

    for(i = bmp_start_pos; i < bmp_size; i = i + 3) begin
        red_i = bmp_data[i+2];
        green_i = bmp_data[i+1];
        blue_i = bmp_data[i];

        #(`clk_period);
        done_i = 1'b1;
    end

    #(`clk_period);
    done_i = 1'b0;

    #(`clk_period);
    writeBMPFromGrayscale;

    #(`clk_period);
    $finish;
end
```

It is necessary to reset all signals using system reset and also dump the waveforms to a .vcd file for debugging.

5. Sobel Operator Analysis

The sobel operator uses two 3x3 kernels which are convolved with the original image to calculate the approximations of the derivatives - one for horizontal changes, and one for vertical. If we define A as the source image, and Gx and Gy are two images which at each point contain the horizontal and vertical derivative approximations respectively, the computations are as follows:

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

Where * denotes the 2-dimensional signal processing convolution operation.

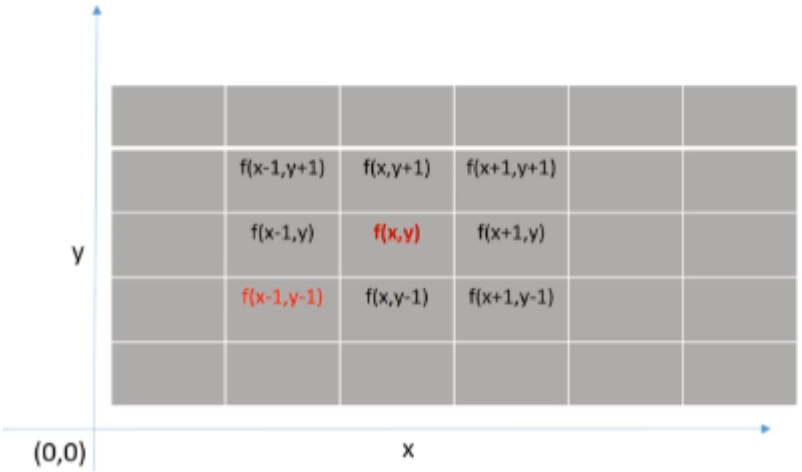
At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude using Pythaorean addition:

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

However, it is too expensive to use square root operator on FPGA, we instead use absolute sum to calculate G:

$$|G| = |Gx| + |Gy|$$

To compute |G| for each pixel, we need to form a 3x3 window where the given pixel is the center in order to compute the Gx and Gy. The following diagram shows the pixel layout for the center pixel f(x,y):

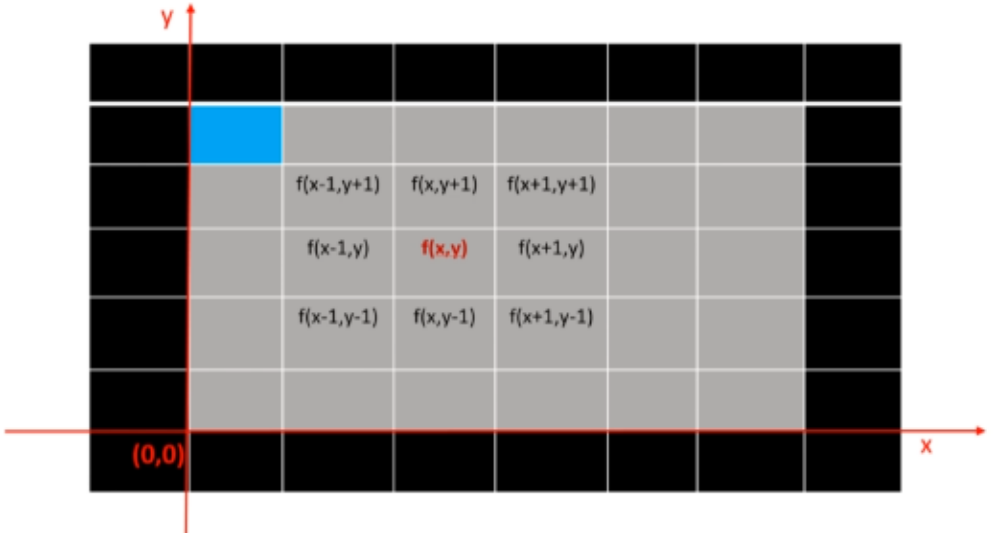


Using the neighbor pixels we can compute Gx and Gy:

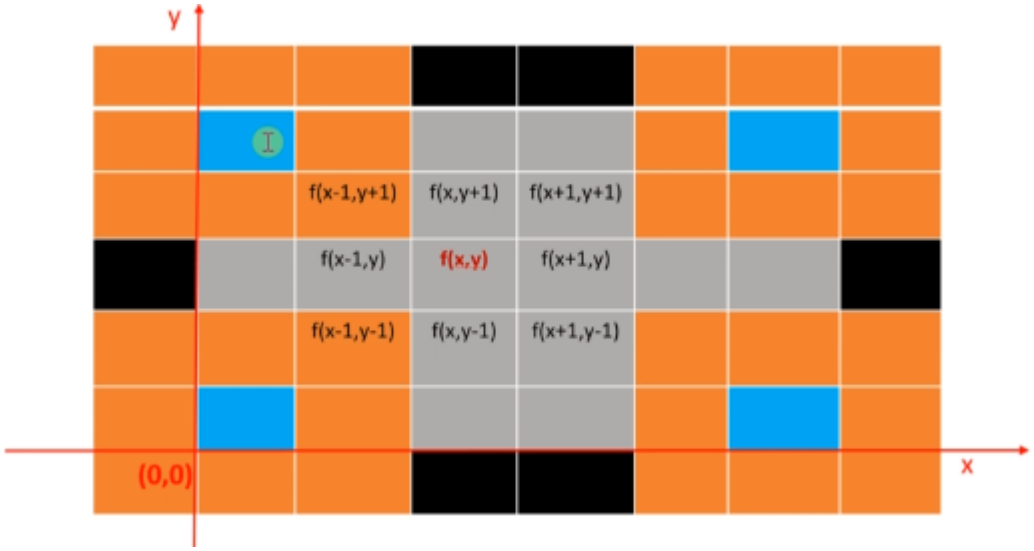
Gx = f(x-1, y-1) * 1 + f(x-1, y) * 2 + f(x-1, y+1) * 1
f(x+1, y-1) * (-1) + f(x+1, y) * (-2) + f(x+1, y+1) * (-1)

Gy = f(x-1, y+1) * 1 + f(x, y+1) * 2 + f(x+1, y+1) * 1
f(x-1, y-1) * (-1) + f(x, y-1) * (-2) + f(x+1, y-1) * (-1)

It is however a bit tricky to compute Gx and Gy for edge pixels because of inadequate neighbouring pixels, for this we can fill the missing neighbouring pixels to 0:

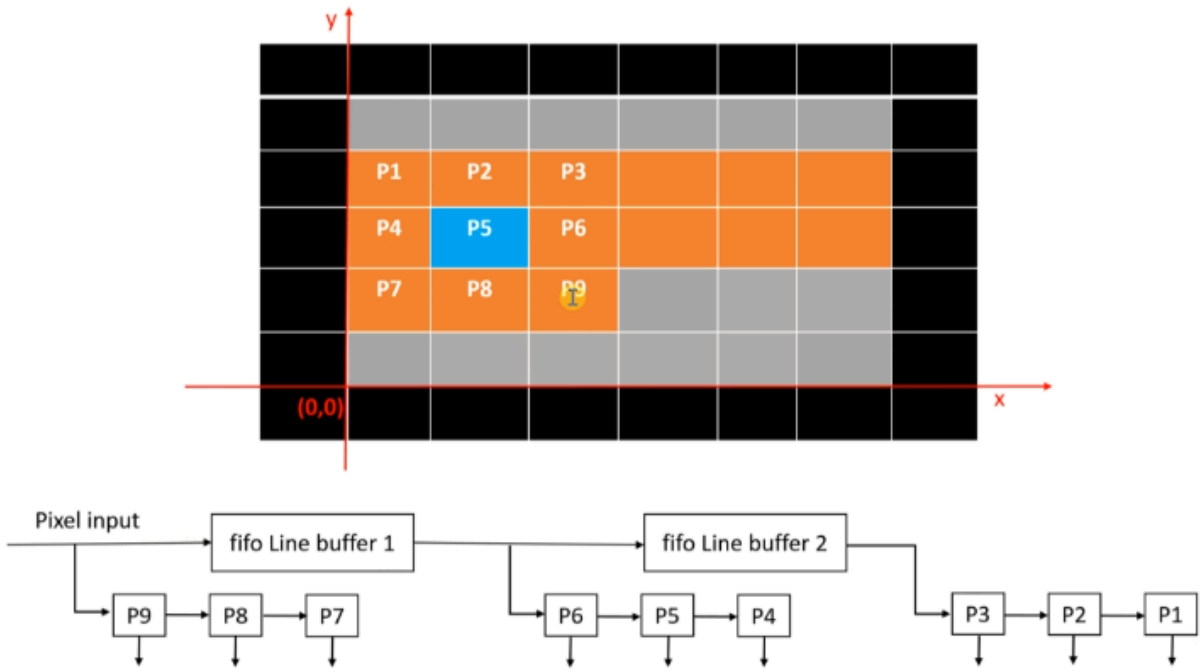


So for the edge points highlighted in blue color in the following diagram, the surrounding orange pixels are needed compute |G| for them:

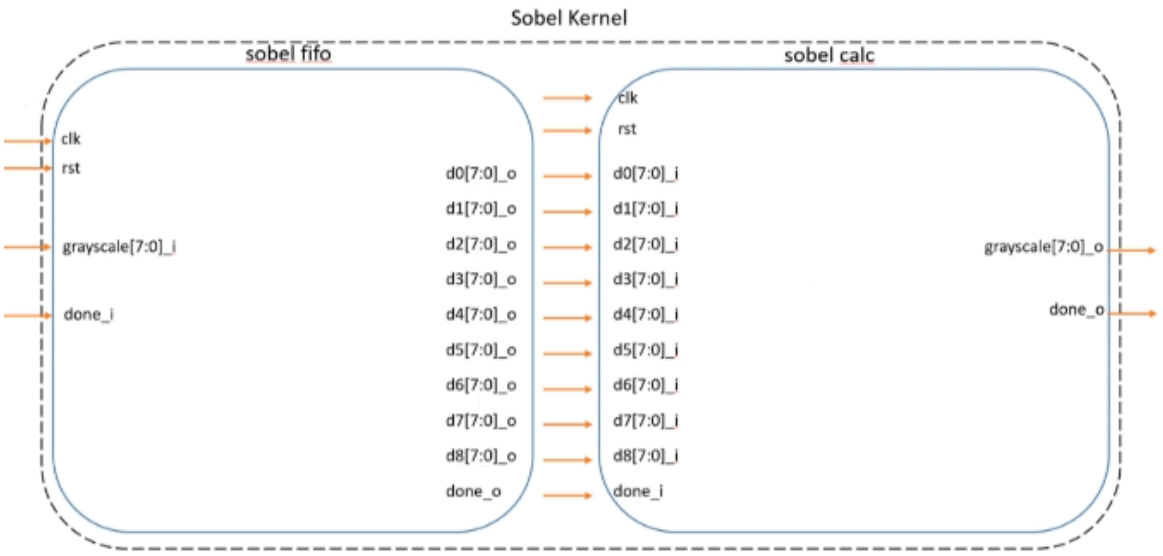


6. Sobel Kernel Interface Analysis

To compute the sobel operator in real time, we need to use FIFO buffers (i.e. line buffers) to store pixels so that we can do the kernel filtering while inputting new pixels. Since this is a 3x3 kernel operator, we need 2 line buffers to store the top 2 rows so that we can use required 9 pixels to compute |G| for the given center pixel. The following diagram shows the line buffer structures in need:

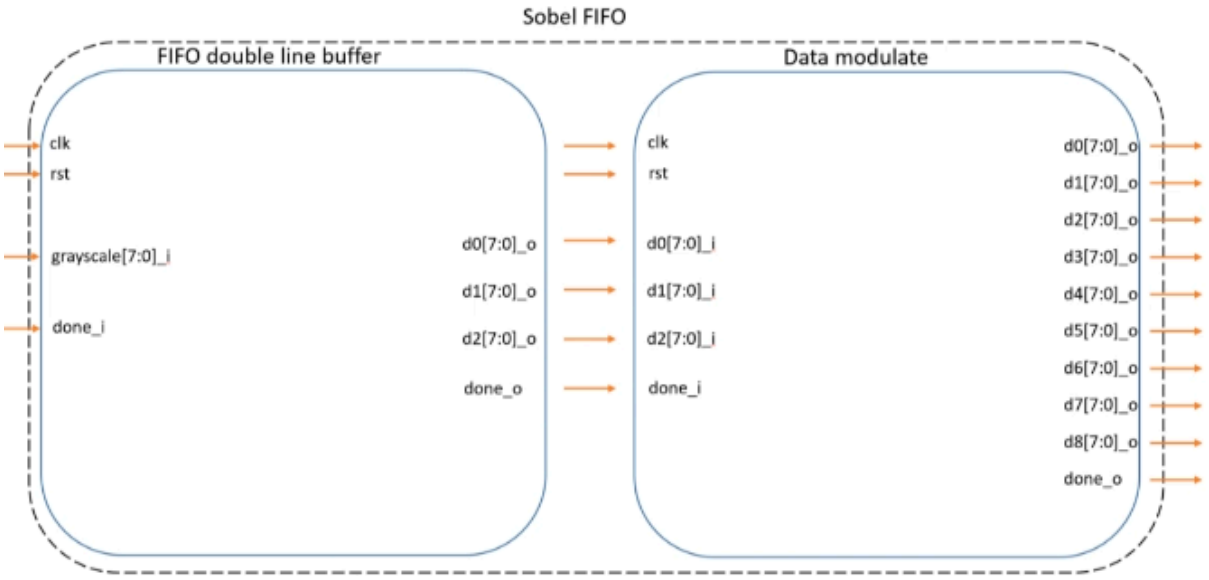


To compute |G| for P5, we need to have P1 - P9 available. For this, we can use two line buffers (i.e. double buffering) to store P1 - P6 so that when P7 - P9 are coming from the input we can compute |G| right away. Note that FIFO line buffer 1 and line buffer 2 do not just store P1 - P6, line buffer 1 stores the entire row to which P4 belongs and line buffer 2 stores the entire row to which P1 belongs.
It is clear now that the sobel kernel module should first buffer input data using line buffers first and then do the sobel convolution, thereby we can break the sobel kernel module into 2 submodules: sobel fifo and sobel calc:

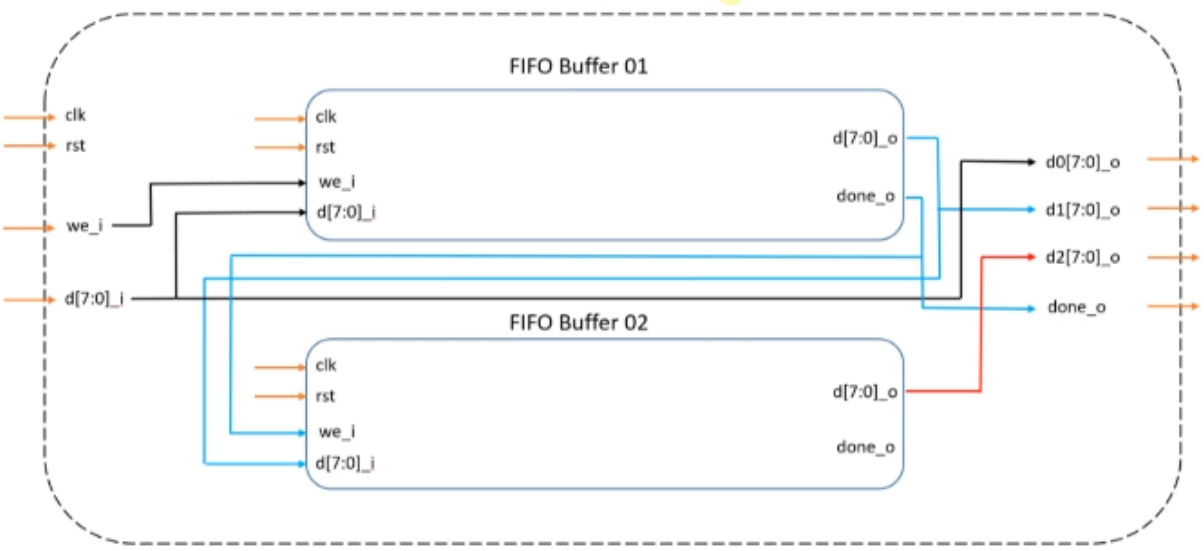


The output ports d0_o to d8_o correspond to the 9 pixels needed for the kernel convolution.

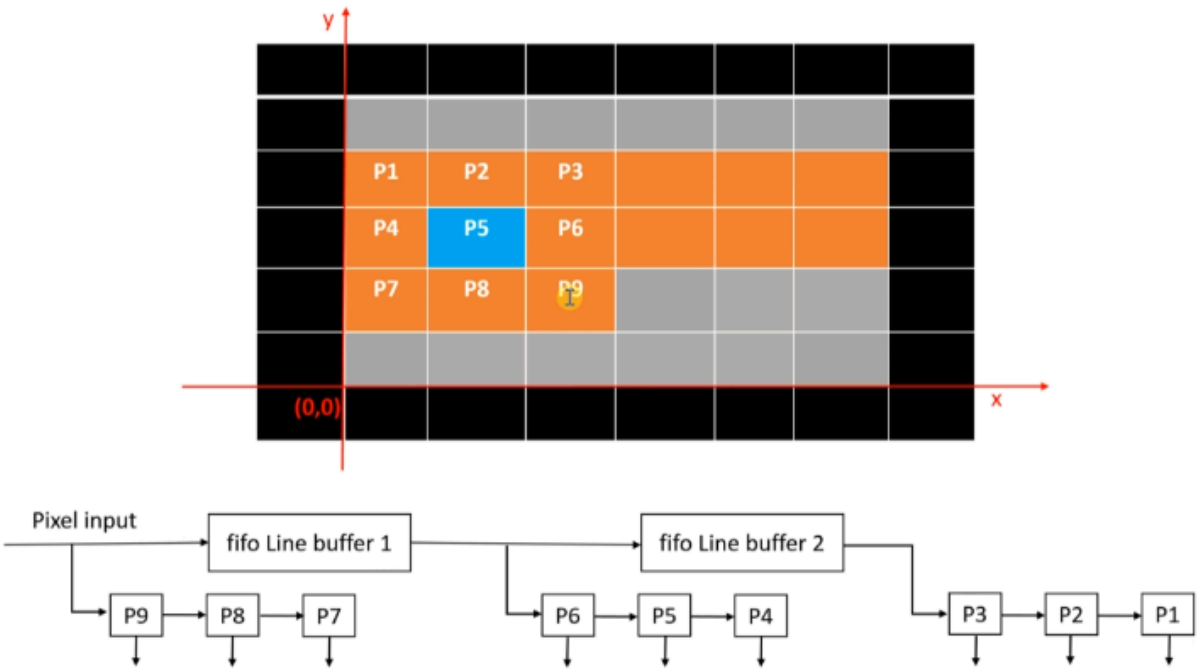
From the line buffer image shown above, we can see that the line buffer each time only outputs 3 pixels instead of 9, so we need a way to modulate the data from the line buffer to form 9-point data output. For this reason, we need to break the sobel_fifo module into 2 submodules: fifo double line buffer module and data modulate module:



Inside the FIFO double line buffer module, we define FIFO buffer 1 and FIFO buffer 2 where buffer 1 forwards data to buffer 2:



It can be seen above that d[7:0]_i connects to d[7:0]_i port of FIFO Buffer 1, d0[7:0]_o connects to d[7:0]_o port and d[7:0]_o from FIFO buffer 2 connects to d2[7:0]_o output port. This is to ensure a single column of data is forwarded to the data modulate module:



From the above diagram we can see that P1,P4,P7 are forwarded together, and this is column 1 for the 3x3 kernel window of P5. Then we have P2P5,P8 forwarded together, etc.

7. FIFO Line Buffer Implementation

A FIFO can be implemented as a circular buffer where we have a writer pointer and a read pointer to handle the writing and reading respectively. Specifically, we need to handle three things:

- Detecting when a FIFO is full. In our case only when the FIFO is full then we can start reading from it. This is because we need to fully buffer one row of pixels before reading according to the line buffering mechanism discussed above.
- Writing: basically update the write pointer and the internal RAM block if write enable is high.
- Reading: read data from the internal RAM block and update the read pointer only if the FIFO is full.

In terms of the internal RAM design, we can use a 1D array of type reg[7:0] to represent. As for the RAM depth, since we are designing the sobel filter to handle images of 128x128 pixels, we can set the array depth to 128 to fill a row.

The buffer fullness design can be done simply by incrementing a counter when the write enable is high and the counter will stay unchanged if we_i is high and the buffer is already full, because overwriting a FIFO does not change its status, which is full in this case.

The fifo interface is constructed according to the FIFO buffer block diagram:

```
module fifo_single_line_buffer (
   input clk,
   input rst,
   input we_i,

   input [7:0] data_i,
   output [7:0] data_o,
   output done_o
);
```

The following is the code snippet handling the above three tasks:

```
//------------------- Handle with iCounter --------------------
   // we can see that the iCounter is used to determine when the FIFO is full.
   always @(posedge clk) begin
      if(rst) begin
         iCounter <= 10'b0;
      end else begin
         if(we_i == 1'b1) begin
            iCounter <= (iCounter == DEPTH ) ? iCounter : iCounter + 10'b1;
         end
      end
   end

   //------------------- Handle with write process --------------------
   always @(posedge clk) begin
      if(rst) begin
         wr_pointer <= 10'b0;
      end else begin
         if(we_i == 1'b1) begin
            mem[wr_pointer] <= data_i;
            wr_pointer <= (wr_pointer == DEPTH - 1) ? 0 : wr_pointer + 10'b1;
         end
      end
   end

   //------------------- Handle with read process --------------------
   always @(posedge clk) begin
      if(rst) begin
         rd_pointer <= 10'b0;
      end else begin
```

```
        // The read pointer is updated after each read once the buffer has been filled.
        if(iCounter == DEPTH) begin
            rd_pointer <= (rd_pointer == DEPTH - 1) ? 0 : rd_pointer + 10'b1;
        end
    end
end
```

Once we complete the single-line FIFO buffer design, we can construct the double-line FIFO buffer by using two single-line buffers and connecting them according to the double-line buffer block diagram:

```
module fifo_double_line_buffer (
    input clk,
    input rst,
    input we_i,

    input [7:0] data_i,
    output [7:0] data0_o,
    output [7:0] data1_o,
    output [7:0] data2_o,
    output done_o
);

wire [7:0] fifo1_data, fifo2_data;
wire fifo1_done, fifo2_done;

assign data0_o = data_i;
assign data1_o = fifo1_data;
assign data2_o = fifo2_data;
assign done_o = fifo1_done;

fifo_single_line_buffer FIFO_SINGLE_LINE_BUFFER_01 (
    .clk(clk),
    .rst(rst),
    .we_i(we_i),

    .data_i(data_i),
    .data_o(fifo1_data),
    .done_o(fifo1_done)
);

fifo_single_line_buffer FIFO_SINGLE_LINE_BUFFER_02 (
    .clk(clk),
    .rst(rst),
    .we_i(fifo1_done),
```

```
    .data_i(fifo1_data),
    .data_o(fifo2_data),
    .done_o(fifo2_done)
);
```

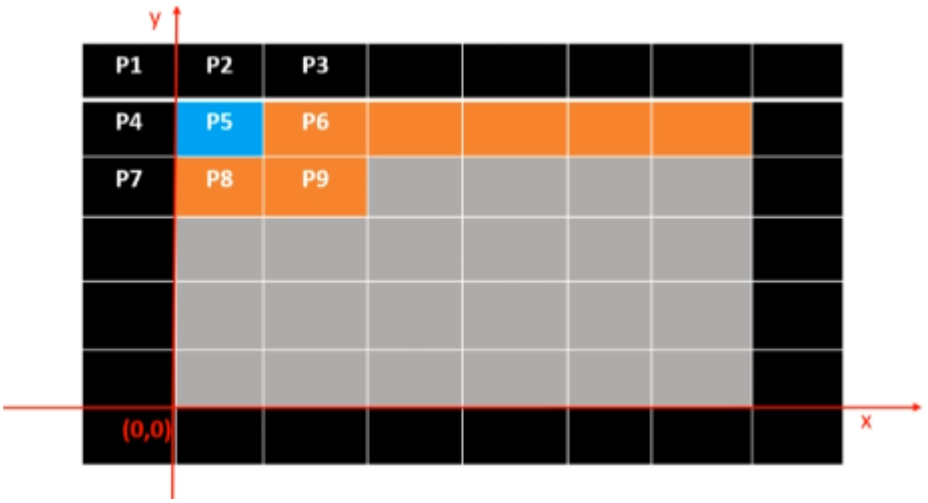
8. Sobel Data Modulation Design

Now we have the double-line buffer delivering a column of pixels every clock cycle, we need to design the sobel data modulate module to structure incoming column pixels to form a 3x3 pixel window for sobel calculation.

To organize input pixels to form a 3x3 kernel window, we need to consider the following 9 cases in terms of center pixel position:

- The top left edge point
- The first row points
- The top right edge point
- The left edge points
- The fully internal points
- The right edge points
- The bottom left edge point
- The bottom edge points
- The bottom right edge point

For example, if the current center input pixel is the top left edge point, then we know P1,P2,P3, P4 and P7 are hardcoded 0 and the rest have input pixel values:



The following code snippet demonstrates this:

```
//----pos 1---- the top left edge point
        if(iRows == 0 && iCols == 0) begin
            d0_o = 0;
            d1_o = 0;
            d2_o = 0;
            d3_o = 0;
            d4_o = data4;
            d5_o = data5;
            d6_o = 0;
            d7_o = data7;
            d8_o = data8;
```

Each edge case has to be handled separately so we need a way to identify which case we need to deal with for a given center pixel. To do so, we can use iCols and iRows variables to keep track of the center pixel. Since we are handling a 128x128 image, we can set the upper bound for both iCols and iRows to be 128:

```
localparam ROWS = 128;
localparam COLS = 128;
```

And we increment both variables when all 9 inputs are available, since that represents we are done with modulating the current kernel window. To generate the ready signal for incrementing iCols and iRows, we can use iCounter to count the number of data columns received so far. When iCounter reaches 2 from 0 then we can keep signaling iCols and iRows to increment since data will come continuously as a data stream.

```
//------------------------handle with iCounter-----------------
// the iCounter is used to track the number of columns received
// so far. We can only modulate data when all 9 pixels (i.e. 3 columns)
```

```
// are available.
always @(posedge clk) begin
    if(rst) begin
        iCounter <= 8'b0;
    end else begin
        if(done_i == 1'b1) begin
            iCounter <= (iCounter == 2) ? iCounter : iCounter + 8'b1;
        end
    end
end
```

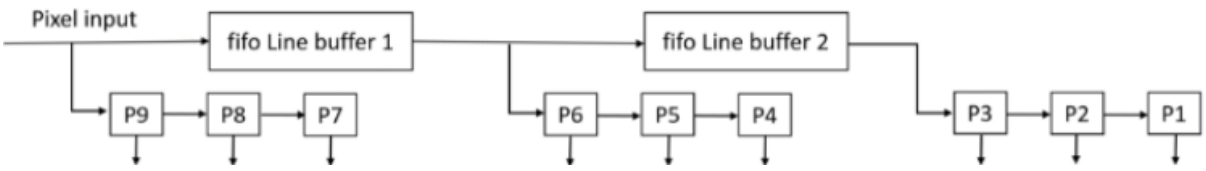
Once iCounter reaches 2 we can set the ready signal to high. The ready signal here can be represented by done_o as it represents the current 3x3 kernel window is modulated:

```
assign done_o = (iCounter == 2) ? 1'b1 : 1'b0;
```

With done_o defined, we can implement a always block to update iCols and iRows whenever done_o is set:

```
always @(posedge clk) begin
    if(rst) begin
        iRows <= 10'b0;
        iCols <= 10'b0;
    end else begin
        if(done_o == 1'b1) begin
            iCols <= (iCols == COLS - 1) ? 0 : iCols + 1;
            if(iCols == COLS - 1)
                iRows <= (iRows == ROWS - 1) ? 0 : iRows + 1;
        end
    end
end
```

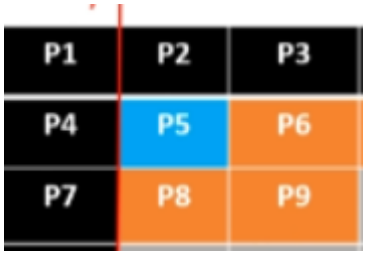
Going back to the center pixel position case study, we use internal registers data0 - data8 to buffer input pixels before updating the output data ports d0_o - d8_o. These 9 internal registers buffer input pixel data by shifting pixels according to the following diagram:



Here, 3 input pixels represent P9, P6 and P3, which correspond to data8, data5 and data2, respectively. So at every clock cycle when done_i is high (i.e. inputs are available and valid), data_1 will be shifted to data0, data2 to data1 and d2_i to data2, and so on and so forth.

```
always @(posedge clk) begin
    if(rst) begin
        data0 <= 8'b0;
        data1 <= 8'b0;
        data2 <= 8'b0;
        data3 <= 8'b0;
        data4 <= 8'b0;
        data5 <= 8'b0;
        data6 <= 8'b0;
        data7 <= 8'b0;
        data8 <= 8'b0;
    end else begin
        if(done_i == 1'b1) begin
            // these 3 lines correspond to the rightmost 3 bits,
            // where d2 comes from the output of fifo line buffer 2
            data0 <= data1;
            data1 <= data2;
            data2 <= d2_i;

            // these 3 lines correspond to the middle 3 bits,
            // where d5 comes from the output of fifo line buffer 1
            data3 <= data4;
            data4 <= data5;
            data5 <= d1_i;

            // these 3 lines correspond to the leftmost 3 bits,
            // where d5 comes from pixel input
            data6 <= data7;
            data7 <= data8;
            data8 <= d0_i;
        end
    end
```

Once we have data0-data8 available, we can assign them to 9 output ports based on the center pixel position. Previously we discussed the case for the top left edge point, similar analysis can be carried out for the rest of cases where zero-filled pixels should be hard coded as 0. Keep in mind that d0_o to d8_o correspond to P1 to P9 in the 3x3 kernel window:



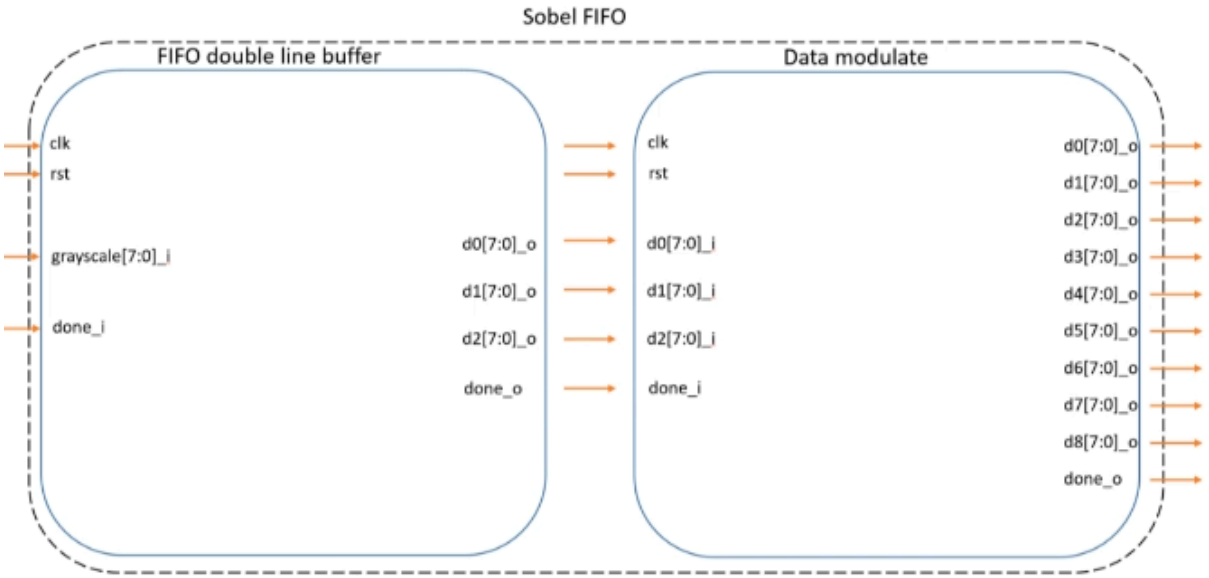
- d0_o : P1
- d1_o : P2
- d2_o : P3
- d3_o : P4
- d4_o : P5
- d5_o : P6
- d6_o : P7
- d7_o : P8
- d8_o : P9

Additionally, each edge case can be determined by checking both iCols and iRows values. For instance, the top left edge point is detected when iCols == 0 && iRows == 0;the first row points are detected when iRows == 1 && iCols > 0 && iCols < COLS - 1.

As the code for the sobel data modulate is large the code will not be shown here but can be found in the code repo.

9. Sobel Data Buffer Implementation

Now we have implemented both double line buffer and sobel data modulate submodules we can construct the sobel data buffer (i.e. sobel FIFO) module according to the block diagram:



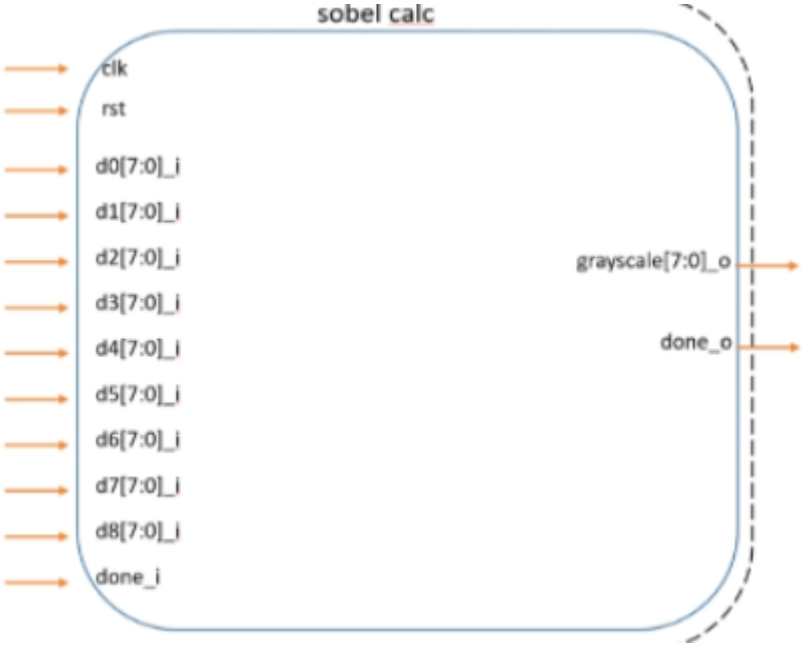
The code is basically wire connections based on the block diagram:

```
module sobel_data_buffer (
    input clk,
    input rst,

    input [7:0] grayscale_i,
    input done_i,

    output [7:0] d0_o,
    output [7:0] d1_o,
    output [7:0] d2_o,
    output [7:0] d3_o,
    output [7:0] d4_o,
    output [7:0] d5_o,
    output [7:0] d6_o,
    output [7:0] d7_o,
    output [7:0] d8_o,
    output done_o
);

wire [7:0] double_line_buffer_data0;
wire [7:0] double_line_buffer_data1;
wire [7:0] double_line_buffer_data2;
wire double_line_buffer_done;

//------------------ fifo double line buffer ----------------
fifo_double_line_buffer FIFO_DOUBLE_LINE_BUFFER (

    .clk(clk),
    .rst(rst),

    .we_i(done_i),
    .data_i(grayscale_i),

    .data0_o(double_line_buffer_data0),
    .data1_o(double_line_buffer_data1),
    .data2_o(double_line_buffer_data2),

    .done_o(double_line_buffer_done)
);

sobel_data_modulate SOBEL_DATA_MODULATE(

    .clk(clk),
    .rst(rst),

    .d0_i(double_line_buffer_data0),
    .d1_i(double_line_buffer_data1),
```

```
    .d2_i(double_line_buffer_data2),
    .done_i(double_line_buffer_done),

    .d0_o(d0_o),
    .d1_o(d1_o),
    .d2_o(d2_o),
    .d3_o(d3_o),
    .d4_o(d4_o),
    .d5_o(d5_o),
    .d6_o(d6_o),
    .d7_o(d7_o),
    .d8_o(d8_o),

    .done_o(done_o)
);

endmodule
```

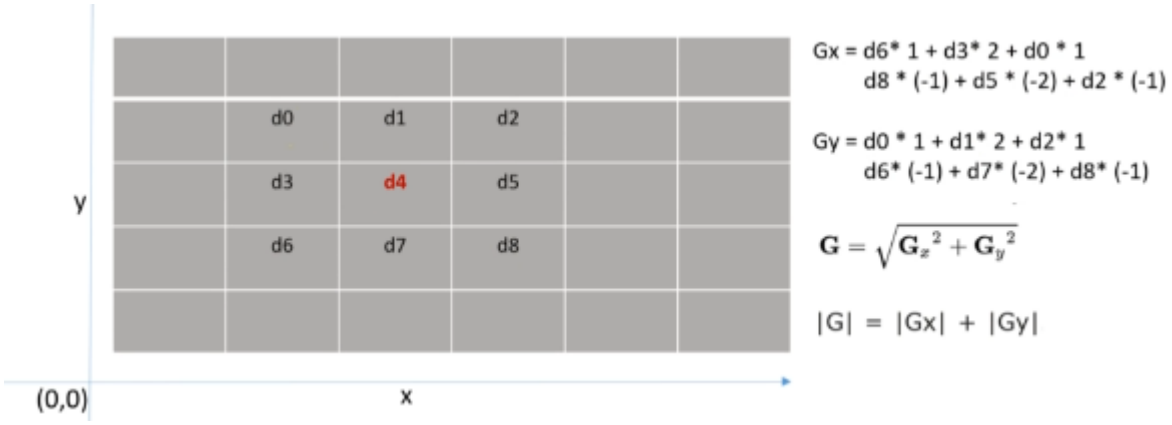
10. Sobel Operator Calculation Implementation

Now we can implement the sobel_calc module:



Within this module we use the 9 input pixels to compute |G| for the centered pixel of these 9 pixels. |G| is basically grayscale_o with 255 being the upper bound.

Recall the sobel operator calculation for a 3x3 kernel window:



Since DSP computation is resource heavy and cycle-consuming, we need to pipeline the calculations into several clock cycles. To do that, we need to refactor the equations to generate intermediate results so that we can perform the calculation in multiple steps, where each step can be done in one clock cycle.

After a few tweak of the given equations, we can land the following equations:

Gx = d6 + d3* 2 + d0 – (d8 +d5 *2 + d2)

gx_p = d6 + d3* 2 + d0

gx_n = d8 + d5 *2 + d2

gx_d = gx_p – gx_n

Gy = d0 + d1* 2 + d2 – (d6 + d7*2 + d8)

gy_p = d0 + d1* 2 + d2

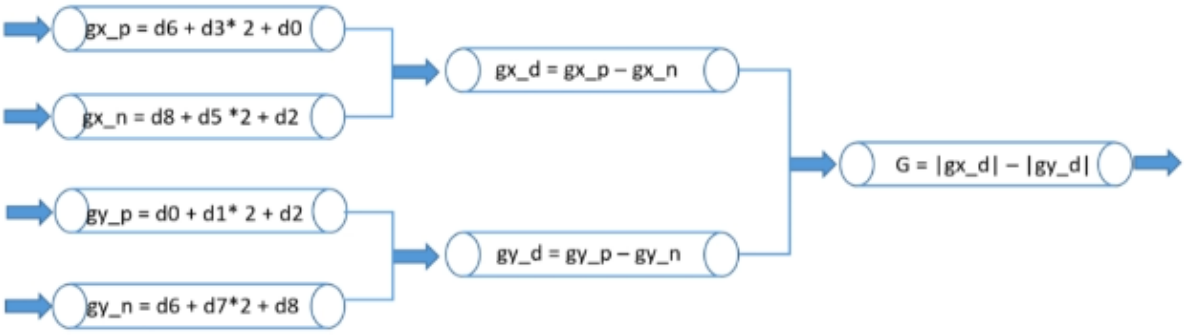
gy_n = d6 + d7*2 + d8

gy_d = gy_p – gy_n

Gx = gx_d ; Gy = gy_d

$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$ |G| = |Gx| + |Gy|

It is clear that we can compute gx_p, gx_n, gy_p, gy_n in one clock cycle, gx_d and gy_d in one clock cycle and G in one clock cycles → we need 3 clock cycles in total to compute G:



```
// calculate gx_p and gx_n
always @(posedge clk) begin
    if(rst) begin
        gx_p <= 0;
        gx_n <= 0;
    end else begin
        gx_p <= d6_i + (d3_i << 1) + d0_i;
        gx_n <= d8_i + (d5_i << 1) + d2_i;
    end
end

// calculate gy_p and gy_n
always @(posedge clk) begin
    if(rst) begin
        gy_p <= 0;
        gy_n <= 0;
    end else begin
        gy_p <= d0_i + (d1_i << 1) + d2_i;
        gy_n <= d6_i + (d7_i << 1) + d8_i;
    end
end

// calculate gx_d and gy_d
always @(posedge clk) begin
    if(rst) begin
        gx_d <= 0;
        gy_d <= 0;
    end else begin
        gx_d <= (gx_p >= gx_n) ? (gx_p - gx_n) : (gx_n - gx_p);
        gy_d <= (gy_p >= gy_n) ? (gy_p - gy_n) : (gy_n - gy_p);
    end
end

// calculate g_sum
always @(posedge clk) begin
    if(rst) begin
        g_sum <= 0;
    end else begin
        g_sum <= gx_d + gy_d;
    end
end
```

Once we have g_sum we can basically update grayscale_o. But instead of directly assigning g_sum to grayscale_o, we can manipulate the g_sum-grayscale_o relationship by using a custom threshold value to determine the limit of g_sum to correspond to 255. This gives us flexibility to adjust the sobel operator based on application scenarios.

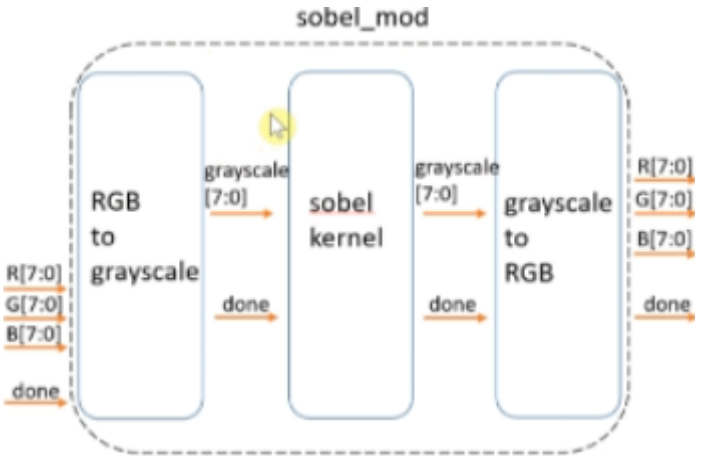
```
// calculate grayscale_o
always @(posedge clk) begin
    if(rst) begin
        grayscale_o <= 0;
    end else begin
        grayscale_o <=(g_sum >= THRESHOLD) ? 8'd255 : g_sum[7:0];
    end
end
```

Finally, we need to output the done_o signal. Since it takes 4 cycles to output grayscale_o we need to delay the input done_i signal by 4 cycles to deliver the done_o signal. We can do this by using a shift register to shift a new done_i signal to right at each clock cycle, so in this way we will have the valid done signal in its MSB data element:

```
reg [3:0] done_shift;

always @(posedge clk) begin
    if(rst) begin
        done_shift <= 0;
    end else begin
        done_shift <= {done_shift[2:0], done_i};
    end
end
```

11. Grayscale to RGB Implementation

Now we reach the final submodule of the sobel_mod module, which is to convert the processed grayscale data back to RGB data.



One quick approach is to repeat the grayscale intensity for each component of RGB. For example, given a grayscale of 120, it translates to RGB (120, 120, 120). We choose this straightforward method for this conversion: we assign R,G,B output registers with the input grayscale value when done is high.

```
always @(posedge clk) begin
    if(rst) begin
        red_o <= 8'b0;
        green_o <= 8'b0;
        blue_o <= 8'b0;
        done_o <= 8'b0;
    end else begin
        if(done_i) begin
            red_o <= grayscale_i;
            green_o <= grayscale_i;
            blue_o <= grayscale_i;
        end

        done_o <= done_i;
    end
end
```

12. Sobel Module Wrapper Implementation

Now we have all submodules implemented, we can create a wrapper module by connecting the following submodules:

- RGB to grayscale
- Sobel kernel
- Grayscale to RGB

```
module sobel_mod (

    input clk,
    input rst,

    input [7:0] cam_red_i,
    input [7:0] cam_green_i,
    input [7:0] cam_blue_i,
```

```
    input cam_done_i,

    output reg [7:0] sobel_red_o,
    output reg [7:0] sobel_green_o,
    output reg [7:0] sobel_blue_o,

    output reg sobel_done_o
);

wire [7:0] sobel_grayscale_i;
wire sobel_grayscale_i_done;

wire [7:0] sobel_grayscale_o;
wire sobel_grayscale_o_done;

rgb_to_grayscale RGB_TO_GRAYSCALE(

    .clk(clk),
    .rst(rst),

    .red_i(cam_red_i),
    .green_i(cam_green_i),
    .blue_i(cam_blue_i),
    .cam_done_i(cam_done_i),
    .grayscale_o(sobel_grayscale_i),

    .done_o(sobel_grayscale_i_done)
);

sobel_kernel SOBEL_KERNEL(

    .clk(clk),
    .rst(rst),

    .grayscale_i(sobel_grayscale_i),
    .done_i(sobel_grayscale_i_done),

    .grayscale_o(sobel_grayscale_o),
    .done_o(sobel_grayscale_o_done)
);

grayscale_to_rgb GRAYSCALE_TO_RGB(

    .clk(clk),
    .rst(rst),

    .grayscale_i(sobel_grayscale_o),
    .done_i(sobel_grayscale_o_done),
```

```
    .red_o(sobel_red_o),
    .green_o(sobel_green_o),
    .blue_o(sobel_blue_o),

    .done_o(sobel_done_o)
);

endmodule
```

13. Sobel Module Simulation

Now we have the sobel_mod module implemented, we can test the module via simulation. We input a 128x128 RGB BMP image as the input and we expect a grayscale image with edges enhanced.

The testbench is very similar to the one used when testing the RGB-to-grayscale module except that we explicitly wait for extra clock cycles before writing the output data to an .bmp file to ensure all sobel operations are completed.

```
// sobel_mod_tb.sv code snippet

initial begin
    rst = 1'b1;
    done_i = 1'b0;

    red_i = 8'b0;
    green_i = 8'b0;
    blue_i = 8'b0;

    $dumpfile("waveforms/sobel_mod_tb.vcd");
    $dumpvars(0, sobel_mod_tb);

    readBMP;

    #(`clk_period);
    rst = 1'b0;

    for(i = bmp_start_pos; i < bmp_size; i = i + 3) begin
        red_i = bmp_data[i+2];
        green_i = bmp_data[i+1];
```

```
        blue_i = bmp_data[i];

        #(`clk_period);
        done_i = 1'b1;
    end

    #(`clk_period);
    done_i = 1'b0;

    for(i = 1; i <= 32; i = i + 1) begin
        #(`clk_period);
    end

    writeBMP;

    #(`clk_period);
    $finish;
end
```

The sobel_mod successfully produced an grayscale edge-enhanced image upon simulation:

Input Image	Output Image
	

This concludes the end of the sobel operator FPGA design.