Lab 4 Report

Adrian Saldana & Noor Fakih

Board 180-23z

**Assignment/Problem Description:**

Lab 4 required us to implement the Sobel algorithm directly in hardware by building a custom accelerator. We were provided with skeleton code, and had to add the logic for the accelerator in order for the program to compute the Sobel convolution with the pixels in the input image, and to add the finite state machines in order to implement the control logic for the design.

**Discussion**:

*Solution/Program Description*

The lab has three major requirements:

1. Finish the implementation of the Sobel convolution for the accelerator in the Verilog file sobel\_accelerator.v
2. Insert the control logic for the accelerator’s state machine and all the other control logic needed inside of the Verilog file sobel\_control.v
3. Be able to successfully synthesize and use the bitstream generated to run our design on our ZedBoard

*Accelerator Core Implementation*

In order to fully implement the accelerator, we had to complete the Verilog code in sobel\_accelerator.v. Sobel\_accelerator.v is where the Sobel convolution is performed. The operands for the x and y convolution were already calculated in the starter code and we had to combine them in a way that faithfully implements the Sobel convolution. We referenced the C version of the Sobel convolution from our previous Lab 2 to make sure that we did not miss any steps. The following code snippets are the only changes we made to complete the implementation of the Sobel convolution and all these changes can be found in sobel\_accelerator.v.

For this process we had to create two regs to temporarily hold the separate x and y convolution sums before we could take the absolute value of them.



We combined the x convolution and y convolution operands separately, and took the absolute value of both.



Then, we calculated the sum of the x and y convolution and ensured that all pixels were at a max value of 255.

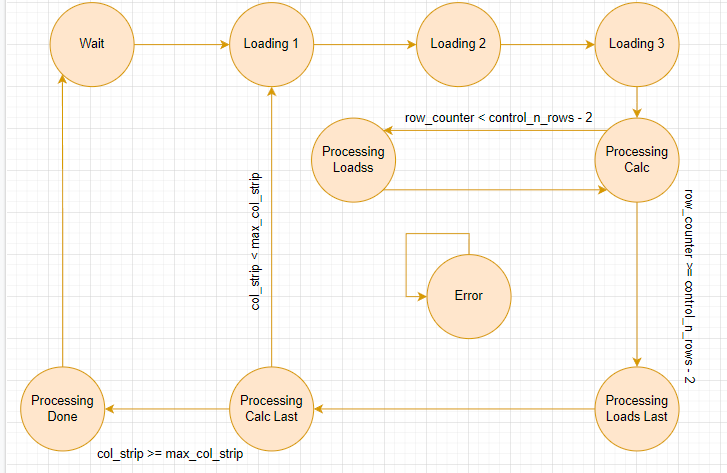


*State Machine Implementation*

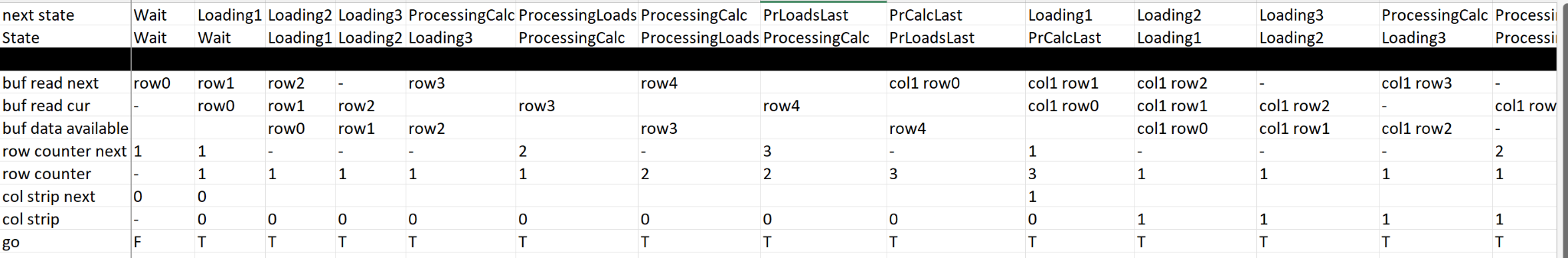
In order to add control logic and add our state machine to our design, we had to heavily edit the sobel\_control.v file. First, we defined some signals that are helpful to use for the rest of the program.

| assign buf\_write\_row\_incr = control\_n\_cols - 'd2; assign next\_col\_strip = col\_strip + `NUM\_SOBEL\_ACCELERATORS; assign max\_col\_strip = control\_n\_cols - (`NUM\_SOBEL\_ACCELERATORS + 'h2); assign pixel\_write\_en[i] = buf\_write\_en; |
| --- |

Next, we implemented the state transitions using the state diagram we created, shown below.



Using our knowledge of the state machine and the desired functionality of our design, we created a timing diagram (pictured below) in order to complete the rest of our design’s implementation.



Row Register Command Generation

Only the states that load in a new row should ever use the row register command `SOBEL\_ROW\_OP\_SHIFT\_ROW, otherwise it should just hold the current rows loaded in.

| always @ (\*) begin  row\_op = `SOBEL\_ROW\_OP\_HOLD;   case (state)  STATE\_WAIT: begin  row\_op = `SOBEL\_ROW\_OP\_HOLD;  end  STATE\_LOADING\_1: begin  row\_op = `SOBEL\_ROW\_OP\_SHIFT\_ROW ; // gets first row - use shift everytime we want a row - EVEN if it is getting a row for the first time setup  end  STATE\_LOADING\_2: begin  row\_op = `SOBEL\_ROW\_OP\_SHIFT\_ROW ; // gets second row  end  STATE\_LOADING\_3: begin  row\_op = `SOBEL\_ROW\_OP\_SHIFT\_ROW ; // gets third row  end  STATE\_PROCESSING\_LOADSS: begin  row\_op = `SOBEL\_ROW\_OP\_SHIFT\_ROW ;   end  STATE\_PROCESSING\_LOADSS\_LAST: begin  row\_op = `SOBEL\_ROW\_OP\_SHIFT\_ROW ;  end  STATE\_ERROR: begin  row\_op = 'hf;  end  default: begin  row\_op = `SOBEL\_ROW\_OP\_HOLD;  end  endcase end |
| --- |

Row Counter

The only state that should ever increment the row counter next is the state “processing calc” since after the initial loading of the first set of 3 rows, it is processing calc who lets processing load know it is ready for the next row. This way by the time it enters the processing load state (which is one clock cycle after processing calc), the row counter flip-flop will have updated the row counter so the processing load will receive the correct next row. State processing calc last sets the row counter to 1 since the program should only ever reach state processing calc last when it is done with the current set of columns and is ready to move onto the next column, where it will have to start over at the first set of rows.

| always @ (\*) begin  // Default behavior is to maintain the current row number.  row\_counter\_next = row\_counter;  case (state)  STATE\_WAIT: begin  row\_counter\_next = 'h1; // at the start of the cols  end  STATE\_PROCESSING\_CALC: begin  row\_counter\_next = row\_counter + 1;  end  STATE\_PROCESSING\_CALC\_LAST: begin  row\_counter\_next = 'h1; // after processing calc last, start over with new col and new rows  end  STATE\_ERROR: begin  row\_counter\_next = 'hf;   end   default: begin  row\_counter\_next = row\_counter;  end  endcase end |
| --- |

Column Strip Counter

State processing calc last is the only state in our design that informs the program what column it should move on to since our program will only ever reach state processing calc when it is done with the current column it is on. It verifies whether or not we are already at the last set of columns to process, if so it will set it to 0 since that means it is done processing the whole image. Otherwise, it will inform the program to start with the next column and go to state Loading 1.

| always @ (\*) begin  // Default behavior is to maintain the current column strip.  col\_strip\_next = col\_strip;    case (state)  STATE\_WAIT: begin  col\_strip\_next = 'h0;  end    STATE\_PROCESSING\_CALC\_LAST: begin  col\_strip\_next = (col\_strip == max\_col\_strip) ? 'h0 : next\_col\_strip; // once at calc last, START OVER  end    STATE\_ERROR: begin  col\_strip\_next = 'hf;   end    default: begin  col\_strip\_next = col\_strip;  end  endcase end |
| --- |

Read Address/Offset Calculation

Only the states that request a new row should ever update its buf\_read\_offset\_next since the read relates to reading the pixels from the input image.

| always @ (\*) begin  buf\_read\_offset\_next = buf\_read\_offset;   case (state)  STATE\_WAIT: begin  if (go) begin  buf\_read\_offset\_next = control\_n\_cols; // get row 1  end else begin  buf\_read\_offset\_next = 'h0; // read row 0 - only happens when its a brand new img  end  end    STATE\_LOADING\_1: begin  buf\_read\_offset\_next = control\_n\_cols + buf\_read\_offset; // read row 2  end    STATE\_LOADING\_3: begin  buf\_read\_offset\_next = control\_n\_cols + buf\_read\_offset;  end    STATE\_PROCESSING\_LOADSS: begin  buf\_read\_offset\_next = control\_n\_cols + buf\_read\_offset;  end    STATE\_PROCESSING\_CALC\_LAST: begin  buf\_read\_offset\_next = control\_n\_cols + buf\_read\_offset;  end    STATE\_PROCESSING\_LOADSS\_LAST: begin  buf\_read\_offset\_next = next\_col\_strip;  end    STATE\_ERROR: begin  buf\_read\_offset\_next = 'hf;  end    default: begin  buf\_read\_offset\_next = buf\_read\_offset;  end  endcase end |
| --- |

Write Address/Offset Calculation

Only the states that need to write should ever increment buf\_write\_offset\_next since writing is related to writing the computed values of the output image from the sobel convolution and that only occurs in states processing calc and processing calc last.

| always @ (\*) begin  buf\_write\_offset\_next = buf\_write\_offset;    case (state)  STATE\_WAIT: begin  buf\_write\_offset\_next = 'h0;  end    STATE\_PROCESSING\_CALC: begin  buf\_write\_offset\_next = buf\_write\_offset + buf\_write\_row\_incr;  end    STATE\_PROCESSING\_CALC\_LAST: begin  buf\_write\_offset\_next = (col\_strip >= max\_col\_strip) ? 'h0 : next\_col\_strip;   end    STATE\_ERROR: begin  buf\_write\_offset\_next = 'h0;  end    default: begin  buf\_write\_offset\_next = buf\_write\_offset;  end  endcase end |
| --- |

Write Enable Generation

Only the states “processing calc” and “processing calc last” should ever set the buf\_write\_en enable signal high since only in processing calc does our design create values that need to be written (i.e. the output image pixel generated by the sobel convolution). Everything else will cause the enable to be set low aka to 0.

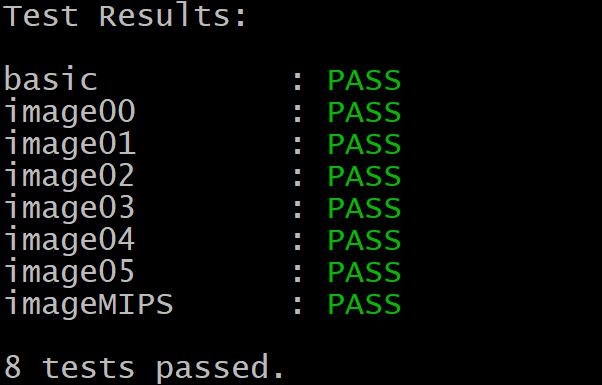
| always @ (\*) begin  buf\_write\_en = 1'b0;  case (state)  STATE\_PROCESSING\_CALC: begin  buf\_write\_en = 1'b1;  end    STATE\_PROCESSING\_CALC\_LAST: begin  buf\_write\_en = 1'b1;  end    default: begin  buf\_write\_en = 1'b0;  end  endcase  end |
| --- |

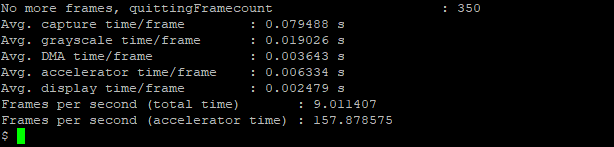
*Implementation Issues, Known Bugs and/or Errors*

Since we were given a skeleton starter code for the lab, any implementation issues that would arise from things like getting the pixels from the input image are already taken care of. Through our testing and analysis of the signal waveforms, we can say there are no bugs that we are aware of in our design.

**Test Description and Results**

Our design for the lab was able to successfully pass all 8 of the provided tests, and successfully synthesize.



****

**Lessons Learned/Epilogue:**

We completed Lab 4 with confidence that we have achieved the learning objectives of the assignment. We built a better understanding of how to implement control logic and utilizing state machines. It also served as a good exercise in creating code with more generality so our design would be flexible enough to accommodate a different number of accelerators.

Our knowledge from EE 108 was very helpful for this assignment as we have previous experience with reading waveforms, making state machine diagrams and timing diagrams, coding in Verilog, etc, and it would have definitely been more challenging if we didn’t. Sneha’s office hours were also very helpful and she gave us a great refresher on timing diagrams and how to properly utilize flip-flops.