FPGA PROJECT

TEAM MEMBERS:

I) Barath S. Narayan (IMT2021524)
II) Pandey Shourya Prasad (IMT2021535)
III) Aditya Garg (IMT2021545)

Viola Jones Algorithm

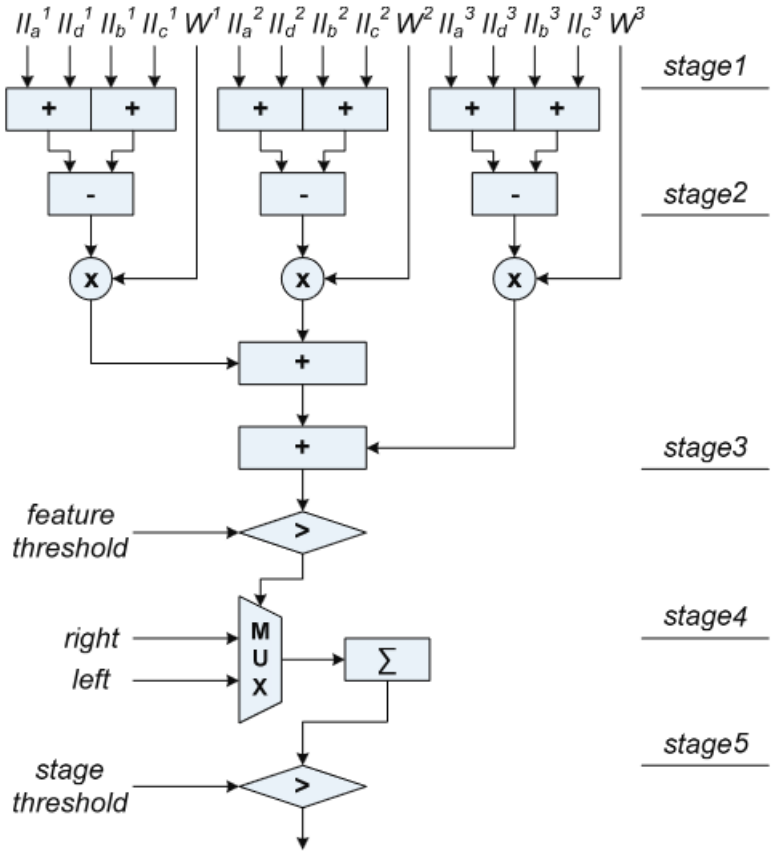
Introduction:

Viola Jones is one of the most widely used methods for real-time object detection, particularly in face detection applications. This algorithm is renowned for its high detection accuracy and efficiency, making it suitable for real-time applications. The algorithm operates through four primary steps:

1. **Haar Feature Selection**: It uses Haar-like features to represent patterns in the image, allowing for efficient differentiation between objects and the background.
2. **Integral Image Calculation**: This step speeds up the computation of Haar features, allowing rapid analysis of regions by computing feature values at any scale in constant time.
3. **AdaBoost Training**: A machine learning approach called AdaBoost is used to select the most relevant features and improve accuracy by creating a strong classifier from a set of weak classifiers.
4. **Cascading Classifiers**: To enhance detection speed, the algorithm arranges classifiers in a cascade, where each stage dismisses regions that are unlikely to contain an object, allowing only probable areas to be further analyzed.

Haar Cascade is an approach where image needs to pass through stages and each stage has different number of features.
There is a kernel of 24x24 which slides over the whole image and the sum of convolution values of each instance is compared with feature threshold value. According to > or <, right or left value is selected (which is given in an .xml file, basically weights) and these are summed up for all the weak classifiers in that particular stage.
The summed up value is then compared with stage threshold value and if it is greater than that then we can say that it that passed that stage. For object detection(Face in our case), all the stages are supposed to be passed for classifying it as detection. Lets suppose we have 10 stages and 7th stage fails, then we need not continue further with 8th, 9th and 10th stage as it has already failed.

Implemented Architecture:



This architecture is split into 5 stages in the diagram:

Stage 1: Integral Image Computation

The top part of the diagram shows an input of various pixel intensities from the integral image, labeled as II_a, II_b, II_c, and II_d. The integral image allows rapid computation of sums over rectangular regions, which are essential for Haar feature calculations.

Stage 2: Haar Feature Calculation

Each block here represents a Haar-like feature, where adjacent rectangular regions are summed, and their differences are computed. For instance, adding and subtracting adjacent regions allows the algorithm to capture patterns like edges or textures within the image. The multiplication by weights W^1, W^2, and W^3 scales the feature values.

Stage 3: Summing the Haar Features

The outputs of the Haar feature calculations are then summed together. This aggregation gives a composite value representing the response of multiple Haar features for a particular region in the image.

Stage 4: Feature Threshold Comparison

In this stage, the combined Haar feature value is compared to a predefined feature threshold. If the feature value exceeds this threshold, it is more likely to represent a positive detection. The MUX chooses between two paths, either “left” or “right”, based on the threshold comparison. If the feature value is less than the threshold, the left value is considered, else, the right value is taken.

Stage 5: Stage Threshold Comparison

Finally, the summed value passes through another thresholding step. Here, the stage threshold determines if the classifier should proceed with additional stages. Only regions passing all threshold checks across all stages will be classified as positive detections.

The code:

The “multi_stage_classifier” module takes a clock input (clk) and uses it to perform classification across multiple stages of Haar feature-based weak classifiers.

Our code has been written in such a way, that the same code can be implemented across various stages.

```
//    output reg signed [27:0] stage_threshold
    reg done;
    wire signed [19:0] sum;
    reg signed[27:0] sumi;
    wire signed [17:0] outi;
    reg pass;
    reg [4:0] i;
    reg [4:0] cycles;
    reg signed [27:0] stage_threshold;
    reg [4:0] num_features[2:0];
    reg [4:0] num_features_stage;
    reg [1:0] stage_number;
//    reg [27:0] stage_threshold;
    reg [3:0] latency_end,latency_start;
    wire [95:0] douta1,douta2,douta0;
    wire [95:0] dina1,dina2,dina0;
    reg address1,address2,address0;
    reg [4:0] address_counter;
    reg wea,ena;
//    reg [15:0] a[2:0],b[2:0],c[2:0],d[2:0],w[2:0];
    wire [15:0] a[2:0],b[2:0],c[2:0],d[2:0];
    wire signed [15:0] w[2:0];
    wire signed [17:0] wa1,wa2,wa3;
//    wire [17:0] outi;
```

Registers and wires:

1) sum: Used to store the summation of feature outputs before the weak classifier classifies
2) sumi: Used to store the accumulated sum over a stage
3) outi: Stores the output of each weak classifier
4) stage_threshold: Threshold for each stage to determine if a candidate region should pass
5) num_features: Used to store the number of features in each stage
6) num_features_stage: Used to store the number of features in the current stage
7) address0, address1, address2: Addresses for accessing memory blocks
8) pass: Signals or flags indicating whether the classification has passed
9) done: Signal or flag indicating whether the stage is complete or not.
10) latency_end: Indicates the number of clock cycles passed spanning from the start of the computations to the generation of the output (outi)
11) stage_number: Indicates the stage we are currently in. Its value only goes up when the “done” signal becomes 1.

In our code, 3 memory blocks are used (blk_mem_gen_0, blk_mem_gen_1, blk_mem_gen_2) to store and retrieve feature data for each region. “douta0”, “douta1”, “douta2” are the outputs from the memory blocks, which contain data for features “a”, “b”, “c”, “d”, and weights “w” for each region.

The “weighted_area” modules “w0”, “w1”, and “w2” compute the weighted Haar features for each region using values from the memory blocks. They take inputs “a”, “b”, “c”, “d”and “w”, and output feature values (wa1, wa2, wa3), corresponding to different feature regions.

The “weak_classifier” module takes the weighted features “wa1”, “wa2”, and “wa3”, and compares them against a feature_threshold. Based on this comparison, it outputs either the “left_value” or “right_value” to “outi” for accumulation. In this case, left_value is taken as -150 and the right value is taken as +150.

The “sum” and “outi” represent the result of evaluating each feature.

Classification Logic:

- The main classification logic is implemented within an always block that triggers on the clock's rising edge.
- **Initialization**: The initial block sets up default values for counters and thresholds. The feature accumulation starts when clk goes high.

- **Feature Accumulation**: “sumi” accumulates feature values (outi) over multiple clock cycles as each feature is processed. The counter “i" tracks the number of features processed.
- **Stage Threshold Comparison**: After all features in a stage are processed, the accumulated value “sumi” is compared with the “stage_threshold”. If it exceeds the threshold, the pass signal is set to 1, indicating the candidate region has passed this stage.
- **Stage Completion**: When a stage is complete (indicated by done), “stage_number” increments, moving to the next stage, and resetting “sumi”, “i”, and counters to prepare for the next set of features.

The code also manages transitions between stages. When “done” signal goes high, the “stage_number” increments, updating “num_features_stage” with the count of features for the new stage. This logic allows the classifier to progress through different stages with varying numbers of features.

Need for Control Signals

The main idea to use this architecture is to ensure the reuse of the hardware for a number of stages by scheduling the data need appropriately. For this reason, a few objectives need to be achieved which are:

1) Correct scheduling of the data to be processed
2) Correct accumulation of the left or right value
3) Exactly stopping the stage after all the weak classifier computations are accumulated.
4) Signalling of the end of a stage

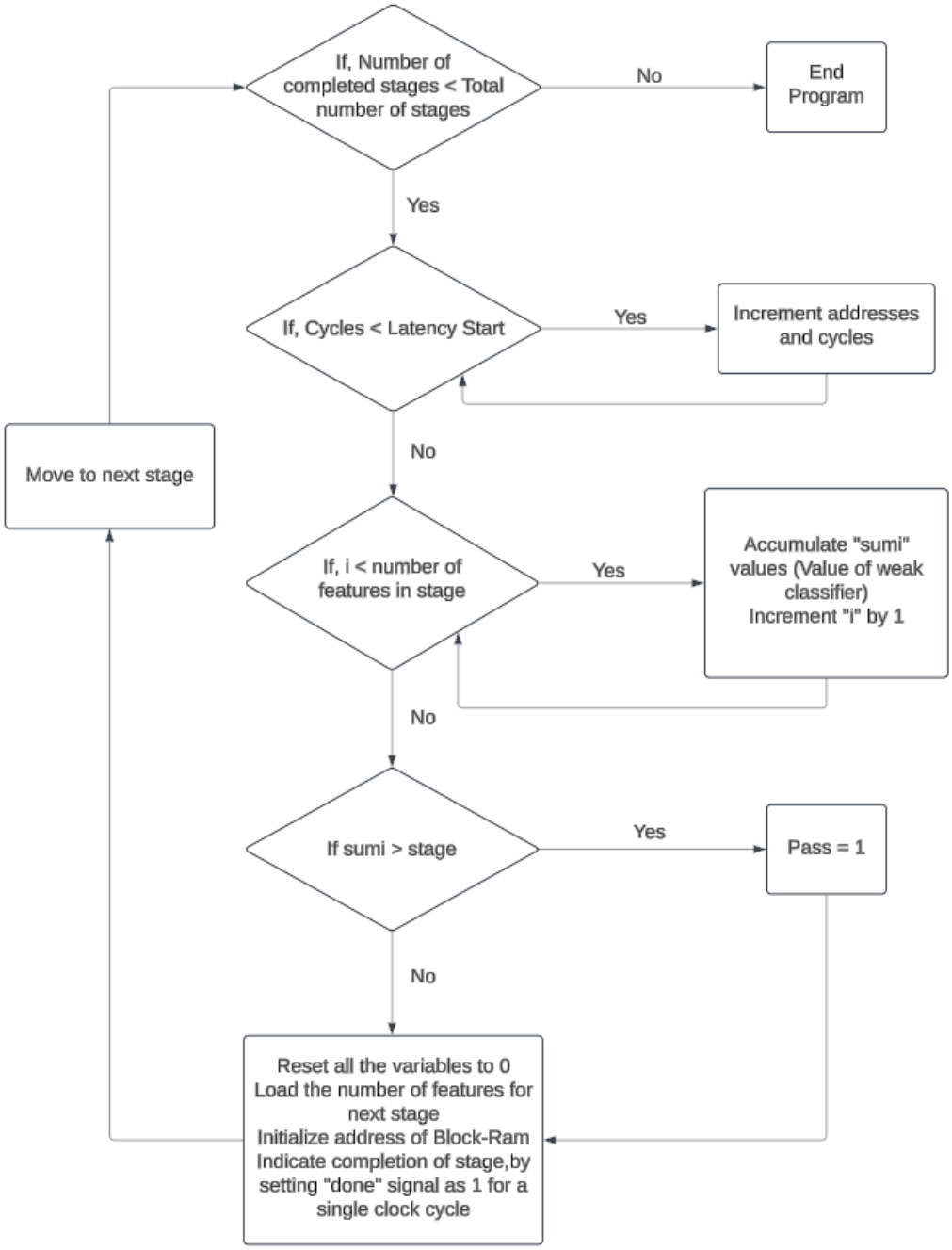
The 1st object is achieved by appropriately storing the data in the block ram and then changing the address accordingly.

The 2nd objective is achieved using the cycles variable and the i variable which accounts for the starting latency and the computation number of the weak classifier result respectively.

The 3rd objective is also completed by correctly aligning the number of correct computations of sumi with the variable i by incrementing i with a delay of latency_start (= 6 for this architecture).

The 4th obective is achieved using the done signal which turns high when i == num_features_stage which is 3 for the 1st stage, 2 for the 2nd stage and 3 for the 3rd stage. The pass signal denotes whether the image actually passes the stage.

A reset signal has been added to ensure that the entire pipeline gets flushed. This can be used by another interfacing controller to reuse the same hardware based on pass or fail of the stage.



Flowchart of Program (Without Reset)

Results of Implementation:

Timing Summary for a 10ns Clock: This indicates that the max frequency of clock which is possible is 1/(10-0.246) ns = 102.52 Mhz

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.246 ns	Worst Hold Slack (WHS):	0.023 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5130	Total Number of Endpoints:	5114	Total Number of Endpoints:	3016

All user specified timing constraints are met.

Intra-Clock Paths - clk - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Des
Path 1	0.246	5	3	block_memory_...am/CLKBWRCLK	w1/out_reg/A[16]	5.831	2.097	3.734	10.0	clk	clk
Path 2	0.300	5	3	block_memory_...am/CLKBWRCLK	w1/out_reg/A[17]	5.774	2.209	3.565	10.0	clk	clk
Path 3	0.354	6	3	block_memory_...am/CLKBWRCLK	w0/out_reg/A[17]	5.718	2.478	3.240	10.0	clk	clk
Path 4	0.404	4	3	block_memory_...am/CLKBWRCLK	w1/out_reg/A[12]	5.674	1.983	3.691	10.0	clk	clk
Path 5	0.438	4	3	block_memory_...am/CLKBWRCLK	w1/out_reg/A[15]	5.633	2.074	3.559	10.0	clk	clk
Path 6	0.461	6	3	block_memory_...am/CLKBWRCLK	w0/out_reg/A[16]	5.615	2.366	3.249	10.0	clk	clk
Path 7	0.468	5	3	block_memory_...am/CLKBWRCLK	w0/out_reg/A[13]	5.604	2.364	3.240	10.0	clk	clk
Path 8	0.473	4	3	block_memory_...am/CLKBWRCLK	w1/out_reg/A[13]	5.601	2.095	3.506	10.0	clk	clk
Path 9	0.476	5	3	block_memory_...am/CLKBWRCLK	w0/out_reg/A[15]	5.592	2.343	3.249	10.0	clk	clk
Path 10	0.533	4	3	block_memory_...am/CLKBWRCLK	w0/out_reg/A[9]	5.539	2.075	3.464	10.0	clk	clk

Utilization Summary:

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Block RAM Tile (50)	DSPs (90)	Bonded IOB (106)	BUFGCTRL (32)	BSCANE (4)
multi_stage_classifier	1742	2604	20	761	1529	213	7.5	3	2	2	
block_memory_0 (blk_mem_gen_0)	0	0	0	0	0	0	1.5	0	0	0	
block_memory_1 (blk_mem_gen_1)	0	0	0	0	0	0	1.5	0	0	0	
block_memory_2 (blk_mem_gen_2)	0	0	0	0	0	0	1.5	0	0	0	
dbg_hub (dbg_hub)	449	741	0	211	425	24	0	0	0	1	
ila_name (ila_0)	1061	1779	20	484	872	189	3	0	0	0	
w0 (weighted_area)	35	0	0	12	35	0	0	1	0	0	
w1 (weighted_area_0)	35	0	0	12	35	0	0	1	0	0	
w2 (weighted_area_1)	35	0	0	13	35	0	0	1	0	0	
wc1 (weak_classifier)	38	37	0	19	38	0	0	0	0	0	

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Block RAM Tile (50)	DSPs (90)	Bonded IOB (106)	BUFGCTRL (32)	BSCANE (4)
multi_stage_classifier	8.38%	6.26%	0.12%	9.34%	7.35%	2.22%	15.00%	3.33%	1.89%	6.25%	25.0
block_memory_0 (blk_mem_gen_0)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	3.00%	0.00%	0.00%	0.00%	0.0
block_memory_1 (blk_mem_gen_1)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	3.00%	0.00%	0.00%	0.00%	0.0
block_memory_2 (blk_mem_gen_2)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	3.00%	0.00%	0.00%	0.00%	0.0
dbg_hub (dbg_hub)	2.16%	1.78%	0.00%	2.59%	2.04%	0.25%	0.00%	0.00%	0.00%	3.13%	25.0
ila_name (ila_0)	5.10%	4.28%	0.12%	5.94%	4.19%	1.97%	6.00%	0.00%	0.00%	0.00%	0.0
w0 (weighted_area)	0.17%	0.00%	0.00%	0.15%	0.17%	0.00%	0.00%	1.11%	0.00%	0.00%	0.0
w1 (weighted_area_0)	0.17%	0.00%	0.00%	0.15%	0.17%	0.00%	0.00%	1.11%	0.00%	0.00%	0.0
w2 (weighted_area_1)	0.17%	0.00%	0.00%	0.16%	0.17%	0.00%	0.00%	1.11%	0.00%	0.00%	0.0
wc1 (weak_classifier)	0.18%	0.09%	0.00%	0.23%	0.18%	0.00%	0.00%	0.00%	0.00%	0.00%	0.0

For 1 module, only 3 block rams were used. For each weighted_area 1DSP was used (in total 3 were used).

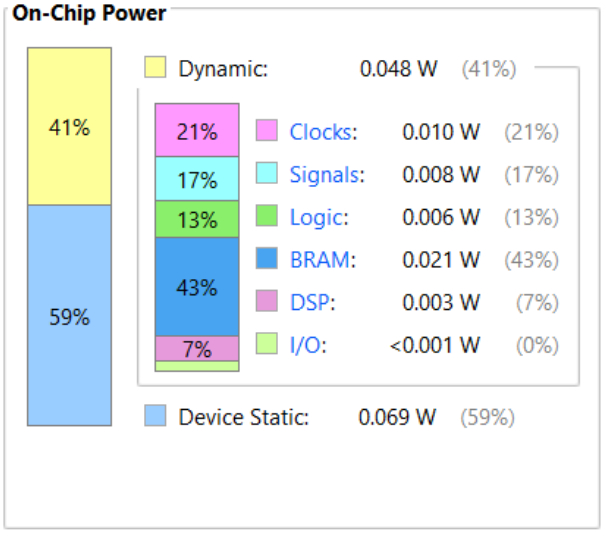
Longest Path delay is 5.831 ns and net delay is 3.734 ns.

Power Summary:

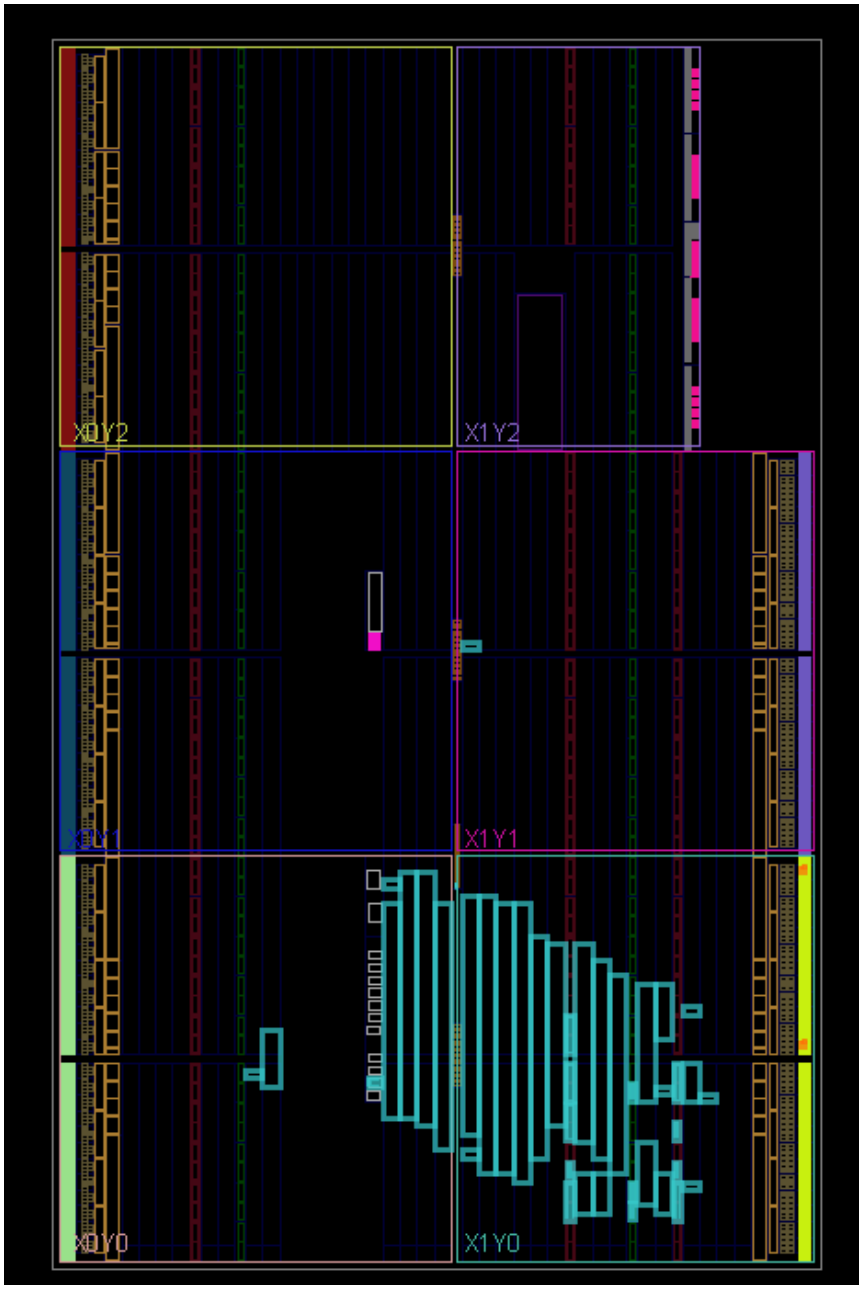
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.117 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**
Thermal Margin:	59.4°C (11.8 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

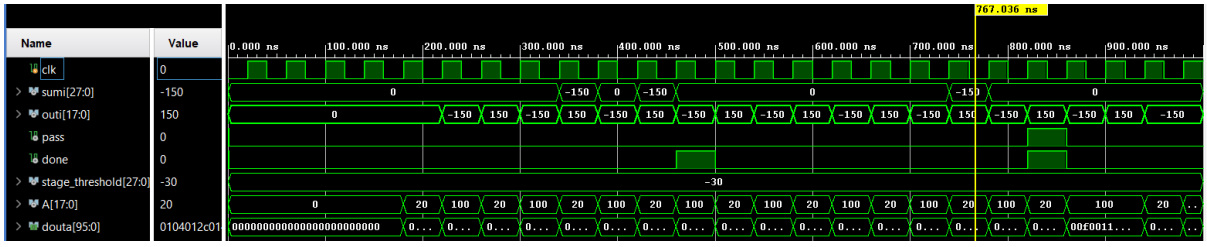


Layout of the Design

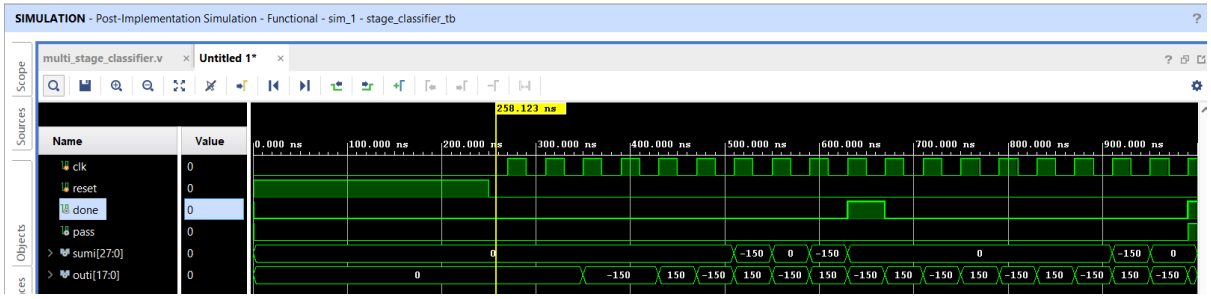


Latency of the implemented design is 4 clk cycles from arrival of data from block ram and a total of 6 clock cycles including latency from block ram.

Without Reset



With Reset

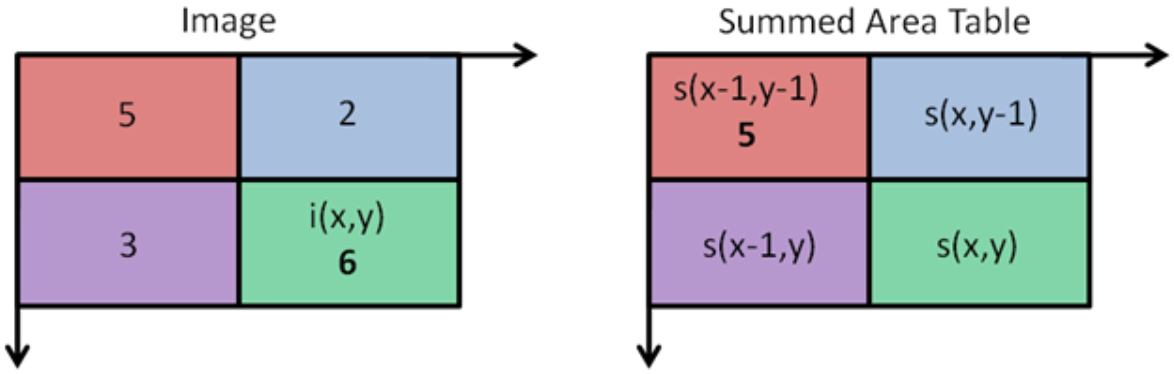


Max throughput achievable depends on the number of weak classifiers in a stage. For a given stage max throughput is 1 computation result (accumulation of result) every cycle once the computation starts. This is assuming perfect scheduling of data is possible for which a separate controller needs to be built.

Haar Cascade Classifier's Python Implementation

Methodology:

We initially calculated Integral image for optimised calculation during the feature comparison.



Integral Image

Code:

```
for i in range(rows):
    for j in range(cols):
        integral_image[i,j]=pixel_array[i,j]
        if i>0:
            integral_image[i,j]=integral_image[i,j]+integral_image[i-
1,j]
        if j>0:
            integral_image[i,j]=integral_image[i,j]+integral_image[i,j-
1]
```

```
        if i>0 and j>0:
            integral_image[i,j]=integral_image[i,j]-integral_image[i-
1,j-1]
```

We then loaded the .xml file and extracted left, right and stage threshold values.

Code:

```
cascade_path="/content/haarcascade_frontalface_default.xml"
doc_Tree = minidom.parse (cascade_path)
width = doc_Tree.getElementsByTagName ("width")[0]
print ("Node Name :",width.nodeName)
width = int(width.firstChild.data)
print ("Node Value :",width)
height = doc_Tree.getElementsByTagName ("height")[0]
print ("Node Name :",height.nodeName)
height = int(height.firstChild.data)
print ("Node Value :",height)
root_node = doc_Tree.documentElement
# Haar feature
rects = root_node.getElementsByTagName ("rects")
print ("root node :",root_node.nodeName)
```

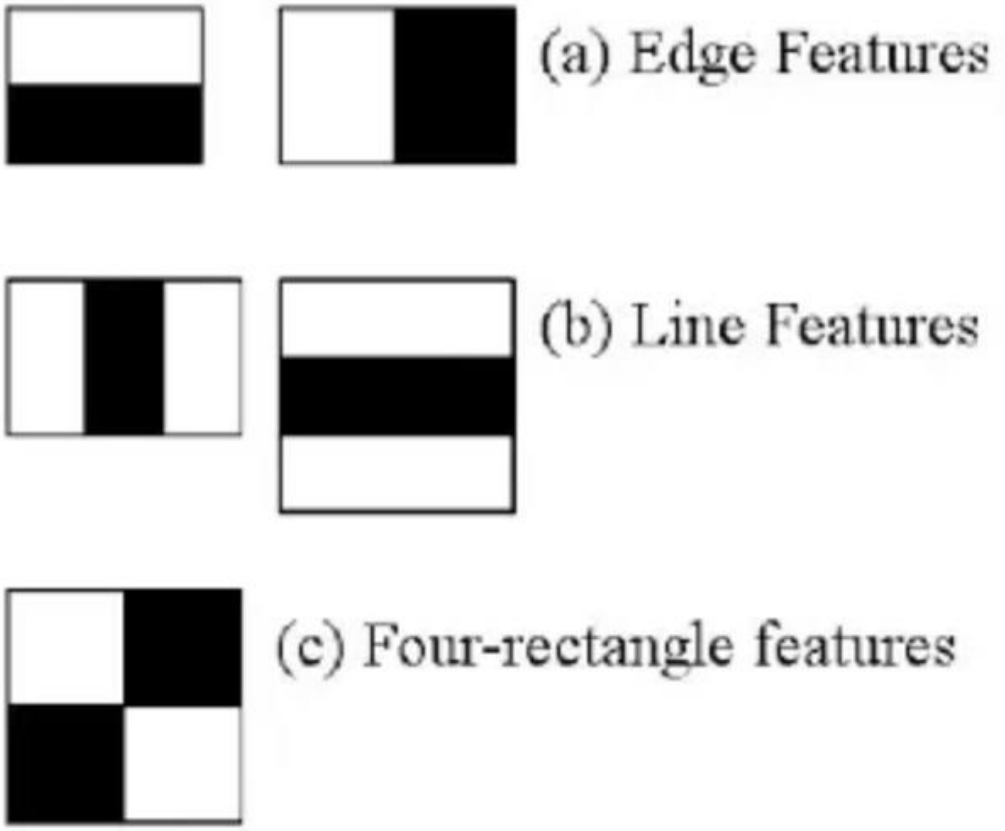
Following this, we made up kernels using the values extracted from the .xml file

Code:

```
for rect in rects:
   kernel = np.zeros((height, width))
   for i, line in enumerate(rect.getElementsByTagName("_")):
       line_list = line.childNodes[0].data.strip().split(" ")
       print(line_list)
       x1, y1, x2, y2 = map(int, line_list[:4])
       # x1, x2 = min(x1, x2), max(x1, x2)
       # y1, y2 = min(y1, y2), max(y1, y2)
       c = float(line_list[4])
       kernel[y1: y1+y2, x1: x1+x2] += c

   feat_mat.append(kernel)
   count += 1

print(count)
```

```
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  2.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
```

Kernel Used for Identifying Face



We can notice horizontal non zero values(our kernel) tends to capture the object.

We further extracted weak and strong classifier values.

Code:

```
weakClassifiers = root_node.getElementsByTagName("weakClassifiers")
stageThresholds = root_node.getElementsByTagName("stageThreshold")
stages_list = []

for weakClassifier, threshold in zip(weakClassifiers, stageThresholds):
   thresholds = float(threshold.childNodes[0].data.strip())
   classifiers = []
   for internalNode, leafValue in zip(
       weakClassifier.getElementsByTagName("internalNodes"),
       weakClassifier.getElementsByTagName("leafValues"),
   ):
       internal_node = internalNode.childNodes[0].data.strip().split("
")
       feat_num = int(internal_node[2])
       feat_thresh = float(internal_node[3])
       leafs = leafValue.childNodes[0].data.strip().split(" ")
       lower_leaf = float(leafs[0])
       upper_leaf = float(leafs[1])
```

```
        classifiers.append([feat_num, feat_thresh, lower_leaf,
upper_leaf])
    stages_list.append([threshold, classifiers])
```

A particular stage value looked like the below image

```
 [[0, -0.031511999666690826, 2.087538003921509, -2.217210054397583],
  [1, 0.012396000325679779, -1.863394021987915, 1.327204942703247],
  [2, 0.021927999332547188, -1.5105249881744385, 1.062572956085205],
  [3, 0.005752999801188707, -0.8746389746665955, 1.1760339736938477],
  [4, 0.015014000236988068, -0.7794569730758667, 1.260841965675354],
  [5, 0.09937100112438202, 0.5575129985809326, -1.8743000030517578],
  [6, 0.0027340000960975885, -1.6911929845809937, 0.44009700417518616],
  [7, -0.018859000876545906, -1.4769539833068848, 0.44350099563598633],
  [8, 0.0059739998541772366, -0.8590919971466064, 0.8525559902191162]]
```

First value denoted the kernel to be used. Second value denotes weak classifier threshold and third and fourth value denotes the left and right value to be taken after comparison with the threshold.

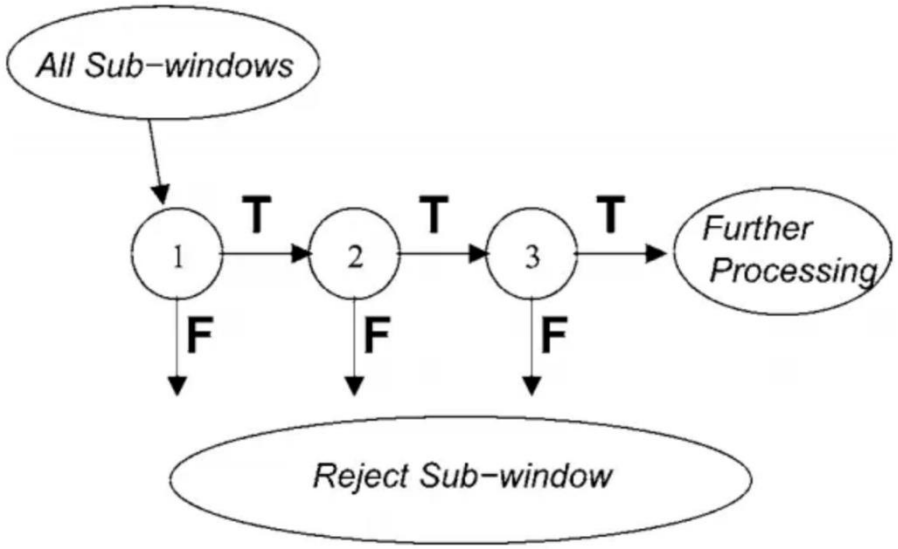
We then implemented the main function which was performing the algorithm.

Code:

```
def checker(x, y):
    N = 24
    pixel_array_copy = [row[y:y + N] for row in img[x:x + N]]
    window = np.array(pixel_array_copy)

    # pixel_array_copy = cv2.equalizeHist(window)

    mean = np.mean(window)
    std_dev = np.std(window)
    pixel_array_copy = (window - mean) / std_dev

    for stage_no in range(len(stages_list)):
        stage_val = 0

        stage_threshold_element =
doc_Tree.getElementsByTagName("stageThreshold")[stage_no]
        stage_threshold_value =
stage_threshold_element.firstChild.nodeValue
        stage_threshold = float(stage_threshold_value)

        for classifier_no in range(len(stages_list[stage_no][1])):
            sum_wi_reci = 0
            feat_mat_idx = stages_list[stage_no][1][classifier_no][0]

            for i in range(N):
                for j in range(N):
```

```
                    sum_wi_reci += feat_mat[feat_mat_idx][i][j] *
pixel_array_copy[i][j]

            # norm = np.std(pixel_array_copy.astype(np.float32))  #
Standard deviation
            norm=1
            wc_threshold = stages_list[stage_no][1][classifier_no][1]
            l_val = stages_list[stage_no][1][classifier_no][2]
            r_val = stages_list[stage_no][1][classifier_no][3]

            rhs = norm * wc_threshold
            lhs = sum_wi_reci

            if lhs > rhs:
                stage_val += r_val
            else:
                stage_val += l_val

        if stage_val < stage_threshold:
            return False

    return True
```

This function performs all the steps in sequential manner. We first normalize the image (24x24 part of the whole image) and convolute it with the kernel. Finally the summed value is compared with the weak classifier threshold value and then left or right value of the feature is added to stage value.
Once all the features calculation is done, we then compare it with the stage_threshold value. If at any stage the stage_val< stage_threshold then we return false because there is no point of comparing the further stages. If it passes then further stages are calculated and when all the stages are done, function returns true which states that the object is found in that 24x24 grid of the image.



We created all the possible 24x24 images from the original image using the code given below.

```
count_true=0
count_false=0
for x in range(0,row-24-1):
 for y in range(0,cols-24+1):
   print(str(x)+" "+str(y))
   val=checker(x,y)
   if(val):
     count_true=count_true+1
   else:
     count_false=count_false+1
 print("True: "+ str(count_true) + " || False: "+str(count_false))
```

Problems Faced:

i) We struggled with extracting values from .xml files and also which value corresponded to what was ambiguous because two different research paper stated two different things.

One of them stated x1,y1,x2,y2 but the other one stated x1,y1,w,h which would change the whole kernel values.

ii) We couldn't figure what was the actual value for the normalization factor.

$$\sum_{i=1}^{k}\left(w_1 \cdot \sum_{u=i+R_1y}^{R_1y+R_1height-1}\sum_{v=j+R_1yx}^{R_1y+R_1width-1} A_{uv}\right) < norm(i,j)\cdot threshold(t)$$

Normalization factor was very important as the image pixel values ranged from 0 to 255 and we had to compare with the threshold values given in .xml file. If normalization was not done properly then wrong comparison result were computed finally leading to wrong answer.

We found two ways to normalize:-

a) Gaussian Normalization
b) Histogram Equalization

We implemented both but still failed to detect face in the image.

iii) We also resized the whole image to 24x24 so as to have our interest of object in 24x24 image and then run our implemented algorithm on it. But it still failed which we feel to be an issue with the normalization factor.

***Verilog and Python code can be found at GitHub.**
[Version 4 at GitHub is final version with reset implemented]

References:

1. Haar Cascade Classifiers in OpenCV Explained Visually.
2. Rapid Object Detection using a Boosted Cascade of Simple Features
3. An Analysis of the Viola-Jones Face Detection Algorithm
4. Face-Recogntion-Detection
5. Haar_Cascade implementation on FPGA