FPGA PROJECT REPORT

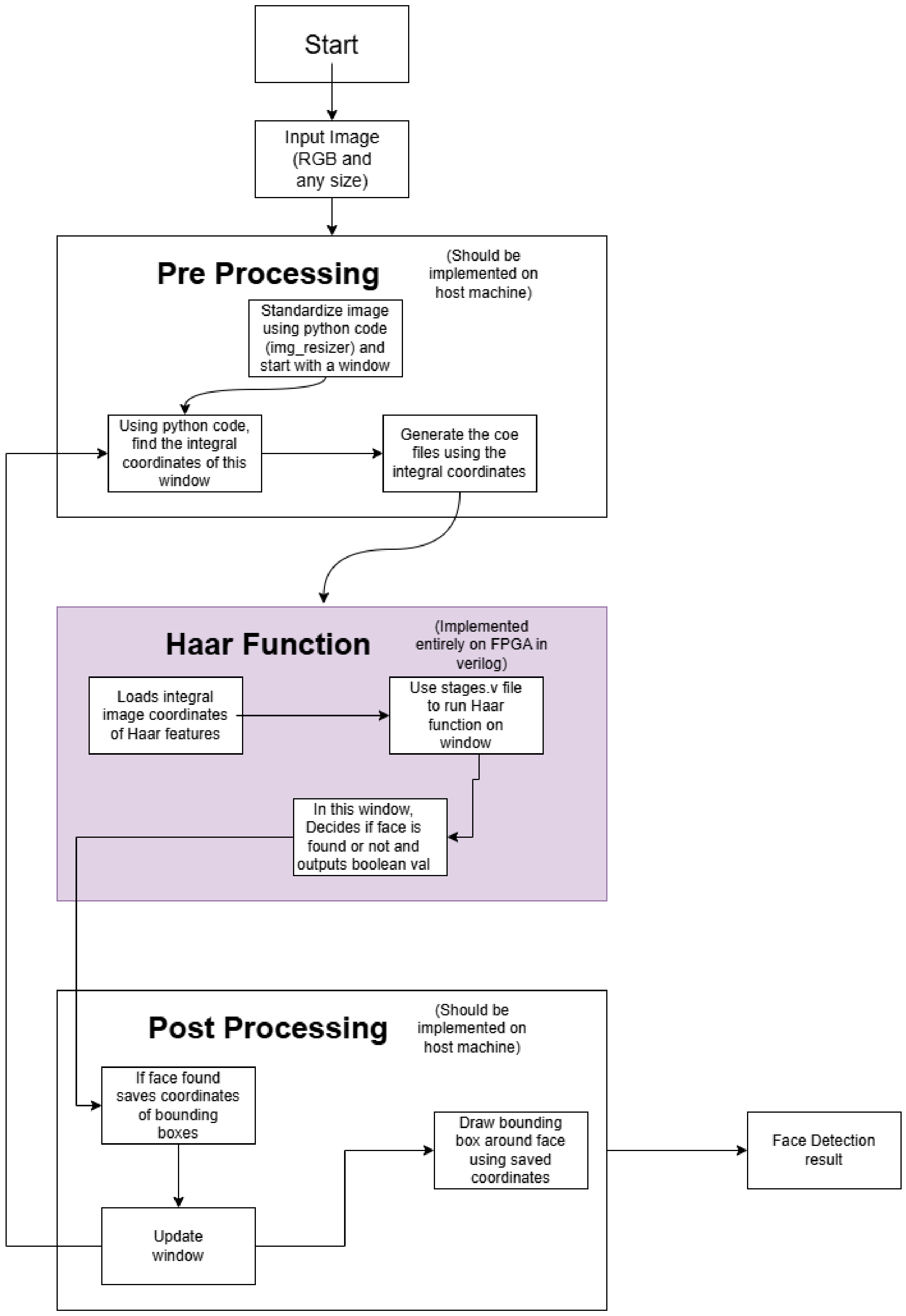
GROUP 1

Submitted to: Prof. Nanditha Rao
Vedant Mangrulkar **IMT2022519**
Abhinav Deshpande **IMT2022580**
Shannon Muthanna **IMT2022552**

Github Repo Link: [Link to Github repo]

Introduction to Haar Cascade Classifier Algorithm

Project Workflow: Aimed Process Diagram



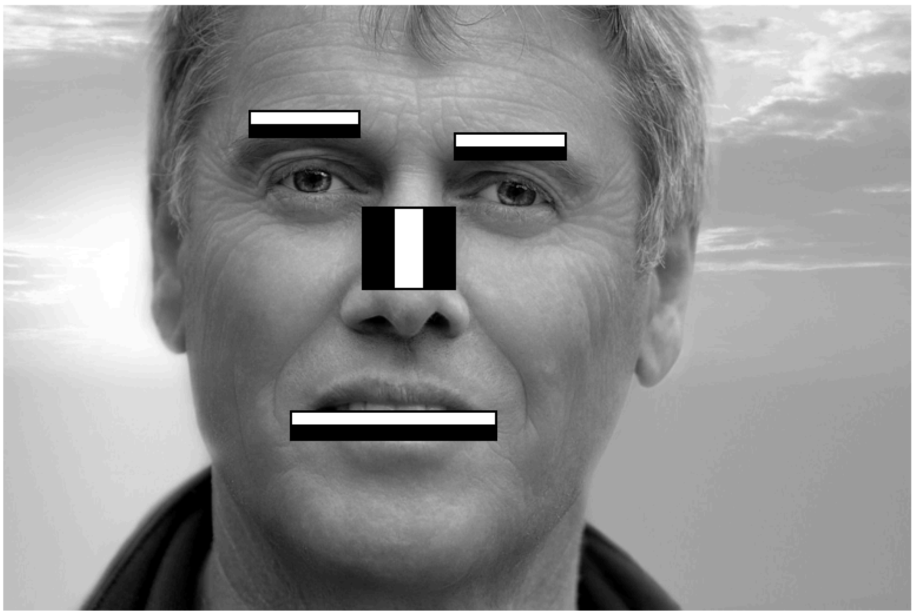
Start
Input Image (RGB and any size)
Pre Processing
(Should be implemented on host machine)
Standardize image using python code (img_resizer) and start with a window
Using python code, find the integral coordinates of this window
Generate the coe files using the integral coordinates
Haar Function
(Implemented entirely on FPGA in verilog)
Loads integral image coordinates of Haar features
Use stages.v file to run Haar function on window
In this window, Decides if face is found or not and outputs boolean val
Post Processing
(Should be implemented on host machine)
If face found saves coordinates of bounding boxes
Update window
Draw bounding box around face using saved coordinates
Face Detection result

The project assigned to us revolves around the face detection algorithm called the Haar Classifier. This algorithm detects the features of the human face and then generalises the output as classifying the face of the human body. This algorithm can be divided into multiple individual tasks:

- Original image -> Gray-Scale image
- Calculating the Integral Image.
- Using the Haar features, iterating them over the face for a specific window of size 24x24.
- Implement the above step for all the stages, Each stage has some x amount of features. Each feature corresponds to a weak classifier. If a window passes all stages then only it is a face region. If it fails in any intermediate stage then we discard that window there and then only proceed to the next stage.
- Accumulate the values of the weak features for a particular stage.

We have implemented the Haar function using verilog on the FPGA. In the pre-processing box, we have implemented image resizing, standardisation part and integral image calculation part successfully. We are getting to know if a face is detected in the image or not by viewing the output of this Haar function. We could not find the correct integral coordinates and store it in coe file. Post processing is yet to be implemented.

Haar features are rectangular regions at a specific location in a detection window. The calculation of Haar features involves summing the pixel intensities in each region along with its weights . These features can be difficult to determine for a large image. This is where integral images come into play because the number of operations is reduced using the integral image.
Each feature is the difference between the sum of pixels in the black region and sum of pixels in the white region. Here features are relative to what we want to detect. If its a face we don't want objects to be considered as a face. The best features are obtained from the Adaboost algorithm. After training adaboost would have found the best threshold for classifying each face pixel as positive or negative. Below diagram has the Haar features of eyebrows, nose and mouth.



Python code:

We tried to implement the algorithm via Python code. This code calculates the integral image and carries out tasks till the calculation of the independent classifier outputs.
We tried to split the Python code into 4 main functions:

- The first function calculates the integral image of the given image.
- The second function parses through the entire XML file and stores it into different variables. (Information extraction).
- The third function calculates the value of the haar features fixed concerning the 24x24 window. This window takes the required integral image pixels for each rectangle of the XML file. It finds the value for the Haar feature of that weak classifier and compares it to that weak classifier threshold and decides to go to the left/right node.
- The fourth function sums up the node values and then compares it to the stage threshold and gives an output of 1/0 stating if the region has passed that particular stage. If the region passes all stages then it is a face region.

The Main of the code is written such that the window moves through the image i.e. like convolution of two kernels of fixed sizes.

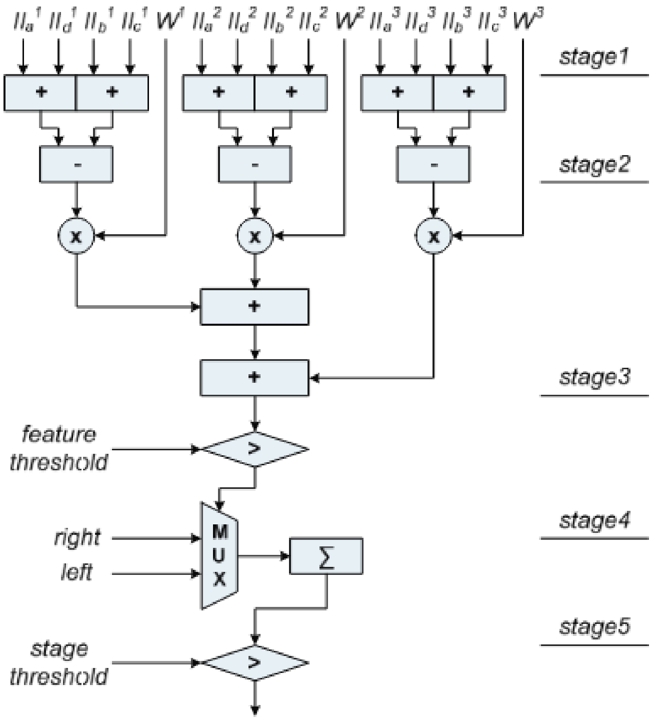
Problems in the Python code:

We did face quite a few issues while implementing the above Python code.
Firstly, understanding this algorithm took a bit of time, because it is a bit complex in terms of the amount of steps getting used to reach the final output.

- Applying features wrt to the window size was difficult as the paper did not mention any reference to this procedure.
- We took the online available xml file and used it by parsing through each line of it as each line contained more than one variable value.
- This xml file did not have any readme attached to it. Hence, it took us some time to understand whether the variables given were coordinates or the width/length of the sides of the rectangle.
- Since the python code was long and was the application of our understanding of the Haar stages code it took us time to write the code.
- After figuring this out, we got stuck at the normalisation part of the code. The values calculated after iterating the image through each feature were always bigger than the weak classifier threshold. Hence, each weak classifier chose the same node leading to not more than 3-6 stages getting successfully passed.
- We tried methods like printing the intermediate calculation and we tried min-max normalisation, standard scaling and calculating the standard deviation of the window and multiplying it with the thresholds.
- Since, none of the features passed all the stages, the matrix storing the features correctly highlighting the facial characteristics was empty.

RTL Implementation

We later switched to implementing the below diagram in verilog with the help of vivado.



II_a^1 II_d^1 II_b^1 II_c^1 W^1 II_a^2 II_d^2 II_b^2 II_c^2 W^2 II_a^3 II_d^3 II_b^3 II_c^3 W^3
stage1
stage2
stage3
feature threshold
right
left
MUX
stage4
stage5
stage threshold

Figure 9. Architecture for performing Haar feature classification.

We implemented this architecture and implemented it for multiple stages. We have a generalised design that works for multiple stages.
Our code consists of three modules: stages, Haar_stages, and WeakClassifier.

WeakClassifiers.v

The WeakClassifier module calculates the weighted intensities for each rectangle. It stores these values in variables and then compares the final weighted intensity of the feature to the weak classifier threshold. Based on this comparison, the code chooses between the right/left node.

This module takes the integral image pixels of all the rectangles along with their weights as inputs. The feature threshold along with the left and right nodes are also inputs. For synchronisation purposes, there is an operation done flag as input. The output of the module is the calculated feature value and a feature value valid flag.

Problems:

1. Miscalculated values due to a mismatch of blocking and non-blocking statements.
2. We were using multiple always blocks which created unnecessary delays and dependencies. Solved it using one always block.
3. Erroneous comparison of feature thresholds and feature values because of signed decimal representation.

Haar_Stages.v

This module reads the inputs from the coe files. These coe files are also generated with the help of a Python code. These coe files are read via block rams.

There are four Block-RAMs in this module. They correspond to rect1, rect2, rect3, and weak classifiers information of the XML file. The rows of this block RAMs correspond to all features and weak classifiers. Stages.v will provide this module with the addresses of the data that it should use for one computation.

This module also takes four inputs which are clock, reset, stage starting address, and the number of weak classifiers of that stage. The output of this module is the final stage decision ie the stage has passed or failed.

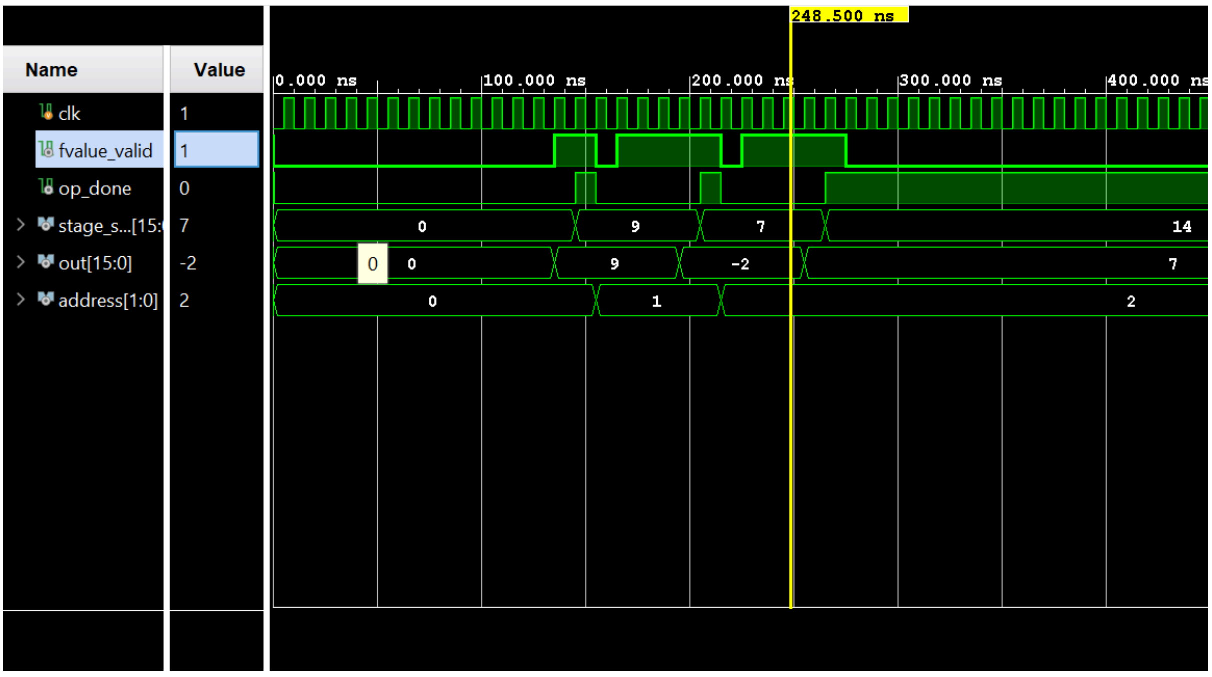
These coe files contain the integral image values along with the thresholds and leaf nodes. The coe files named int_img store all integral image pixels and weights in one location. We represented all our variables in 16 bits thus the data stored in one location is 80 bits (4 pixels and one weight).
This module parses these coe files and stores the data into suitable variables. The WeakClassifier module is called here while passing the inputs to that module.
This module contains the FSM logic which communicates between the WeakClassifier module with the help of signals. The synchronisation signals are feature value valid and op done.
The FSM contains 3 states

- The zeroth state calculates the summation of the feature node values once the WeakClassifier module has stored the node value i.e. valid is one and op done is zero. Once this summation is achieved, the state further changes to state 2 or else stays in the same state.This communication between the 2 modules is achieved using the ‘fvalue_valid’ and ‘op_done’ signals.
- The first state compares the stage threshold value to the summed value of all the weak classifiers only after each weak classifier has completed its calculation i.e. when the address has reached (number of weak classifiers + starting address -1). This is a signed comparison between these 2 values. If not, the address of all the BRAM is increased by one and the state is moved to the second.
- The second state works as a stalling state. It functions on the count variable which ensures that the correct output arrives at the accumulation state. The count is 3

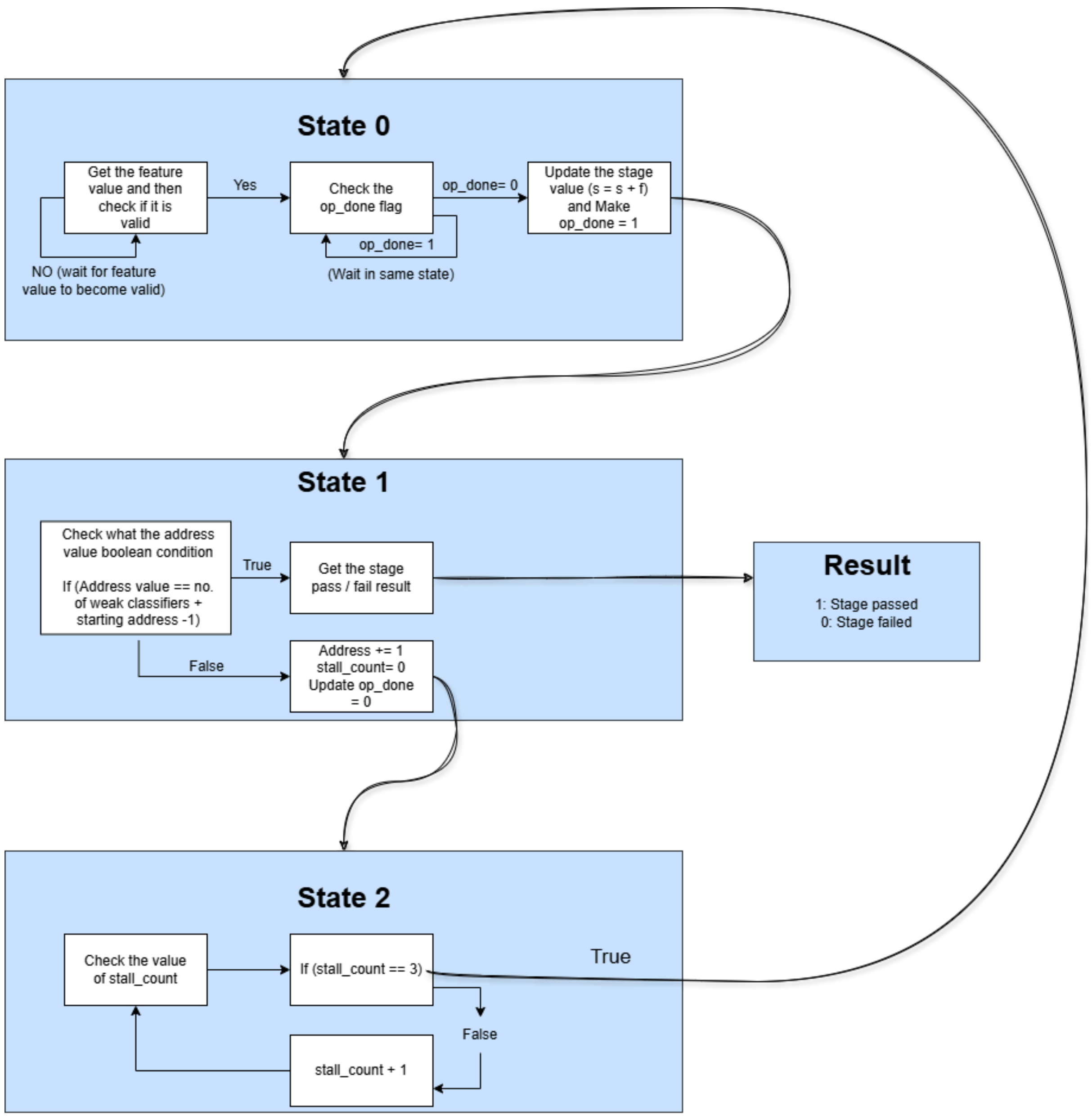
because 2 cycles delay for block-ram and then 1 cycle delay for the output to come so count is 3.

We did face quite a few issue here as well:

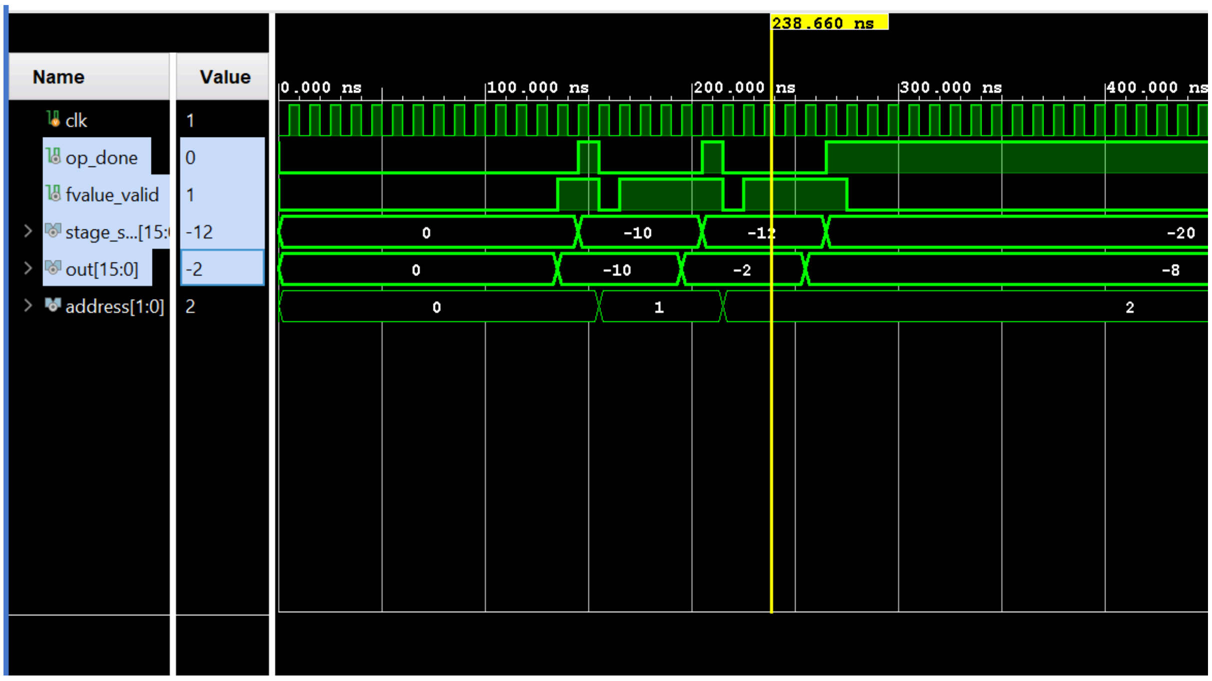
- The block RAM was reading 0 value in the initial few clock cycles. This we resolved by putting a constraint on the inputs that if they are zero ignore them.
- Secondly, the behavioural simulation was working fine, but the Post-Synthesis simulation along with the Post-Implementation simulation were giving wrong results. The simulation showed that both the nodes were being used while calculating the weighted intensity.
  We resolved this issue by adding the second state into our fsm which essentially works as a Stalling state. This state ensures that the correct output arrives at the accumulation state of the fsm.

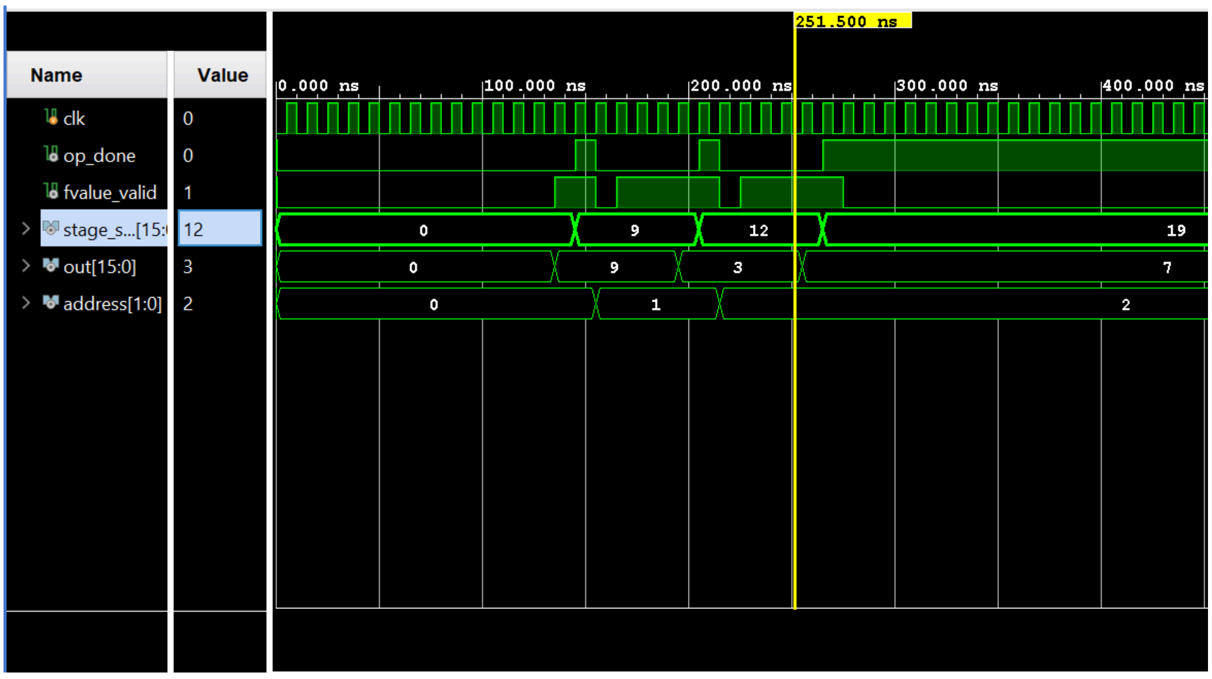


FSM of Stage decision


State 0
Get the feature value and then check if it is valid
Yes
NO (wait for feature value to become valid)
Check the op_done flag
op_done= 0
op_done= 1
(Wait in same state)
Update the stage value (s = s + f) and Make op_done = 1
State 1
Check what the address value boolean condition
If (Address value == no. of weak classifiers + starting address -1)
True
Get the stage pass / fail result
False
Address += 1
stall_count= 0
Update op_done = 0
Result
1: Stage passed
0: Stage failed
State 2
Check the value of stall_count
If (stall_count == 3)
True
False
stall_count + 1

We verified this code with the help of 3 different kinds of inputs, checking the edge cases. Following were the outputs:





Stages.v

The input of this module is a clk and the output of the module is the final haar decision which tells if it is a face or not. This module utilizes a block RAM whose each row consists of information of all stages. Each stage has two things number of weak classifiers and the starting address of each stage. These two things are passed as inputs to the Haar stages module. Using this information Haar stages module can correctly calculated the stage decision using the required weak classifiers data.

The main logic of this module is an FSM which keeps track of how many stages have passed and makes the final classification based on them.

State 0: Waits until any stage returns its stage decision as one. Once stage decision is one it increments the number of stages passed and goes to the next state.If stage decision is zero then remain in this state.

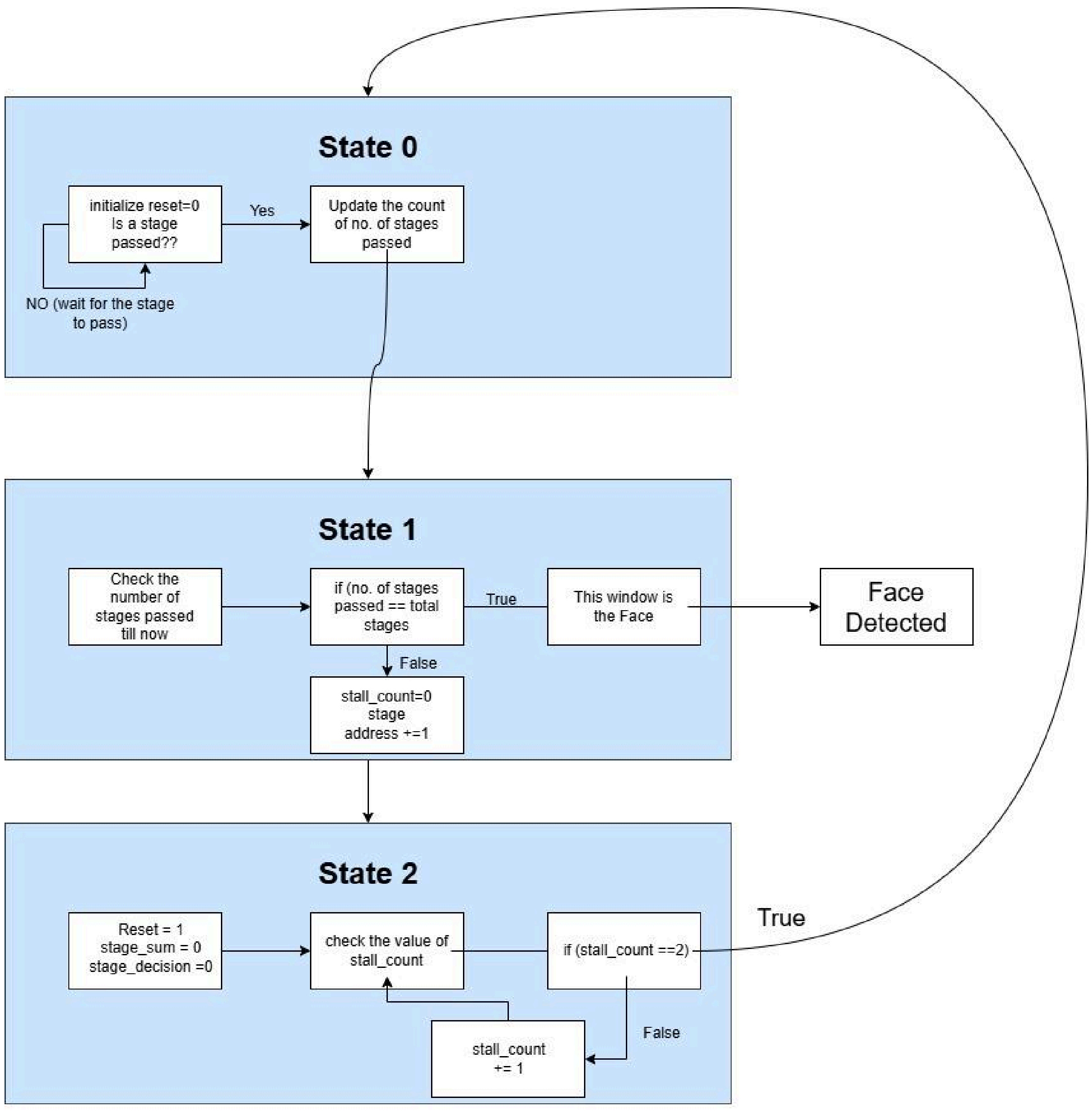
State 1: If the number of stages passed is equal to the total number of stages then the final face decision is one or else increment the stage address which corresponds to the BRAM which contains the data of all stages. Thus here the stage address gets updated only if the previous stage has passed. After updating the address it goes to the stall state because it takes 2 cycles to arrive after the address is updated.

State 2:
In this state the reset is made, one which makes the stage decision and final stage sum go to zero. This reset is made back to zero when it goes back to state 0. The stall state has a counter which counts till two and then goes to state zero.

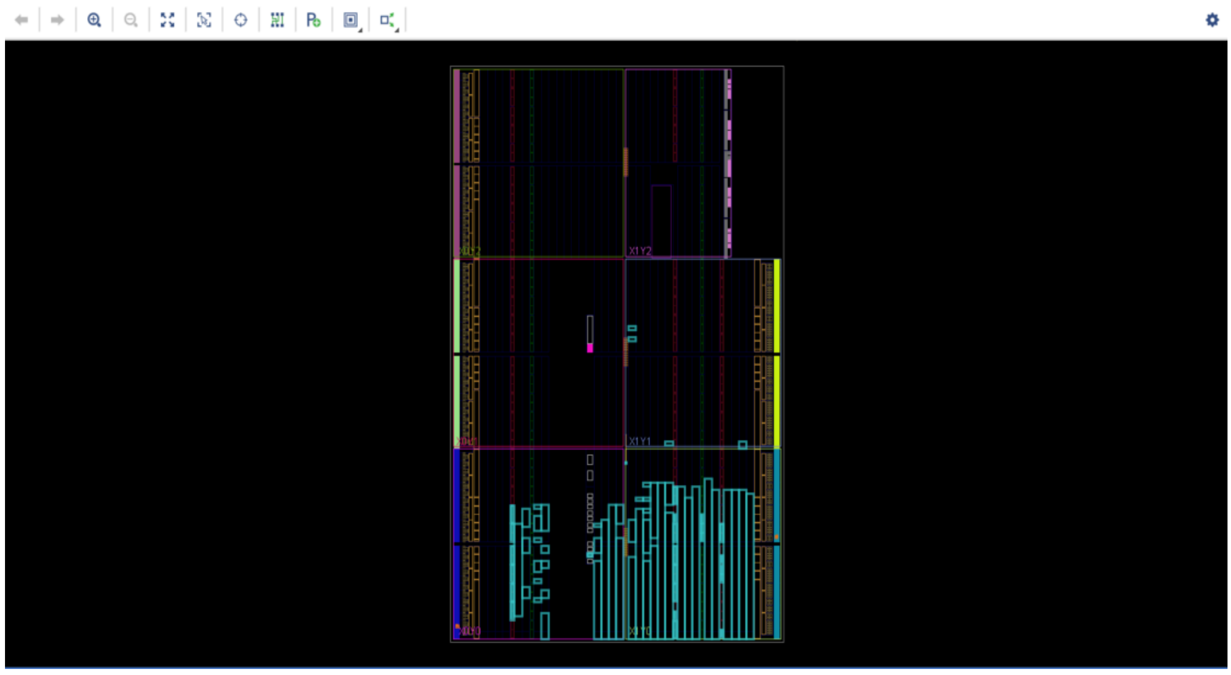
Problems:

1. Had to include reset pins to bring stage decision and final stage sum to zero.
2. Stalls were needed to synchronise the data flow.

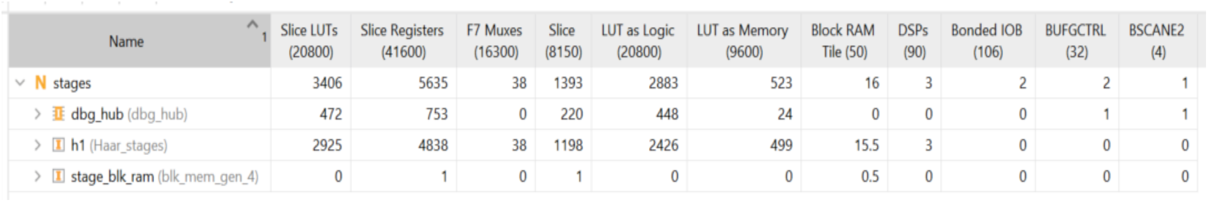
FSM of Stage Decision calculation and Face Detection output



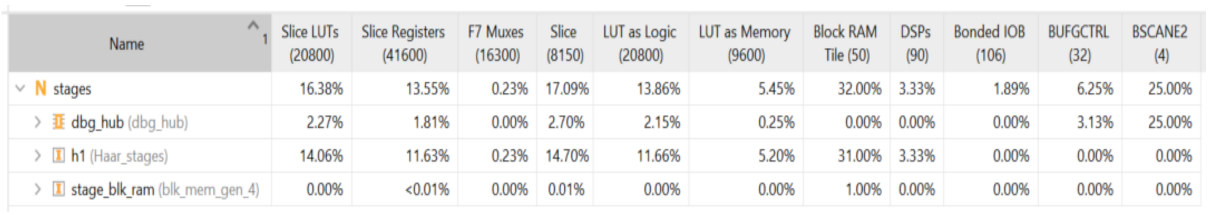
IMPLEMENTED LAYOUT DIAGRAM:



RESOURCE UTILISATION:



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)	BSCANE2 (4)
stages	3406	5635	38	1393	2883	523	16	3	2	2	1
dbg_hub (dbg_hub)	472	753	0	220	448	24	0	0	0	1	1
h1 (Haar_stages)	2925	4838	38	1198	2426	499	15.5	3	0	0	0
stage_blk_ram (blk_mem_gen_4)	0	1	0	1	0	0	0.5	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)	BSCANE2 (4)
stages	16.38%	13.55%	0.23%	17.09%	13.86%	5.45%	32.00%	3.33%	1.89%	6.25%	25.00%
dbg_hub (dbg_hub)	2.27%	1.81%	0.00%	2.70%	2.15%	0.25%	0.00%	0.00%	0.00%	3.13%	25.00%
h1 (Haar_stages)	14.06%	11.63%	0.23%	14.70%	11.66%	5.20%	31.00%	3.33%	0.00%	0.00%	0.00%
stage_blk_ram (blk_mem_gen_4)	0.00%	<0.01%	0.00%	0.01%	0.00%	0.00%	1.00%	0.00%	0.00%	0.00%	0.00%

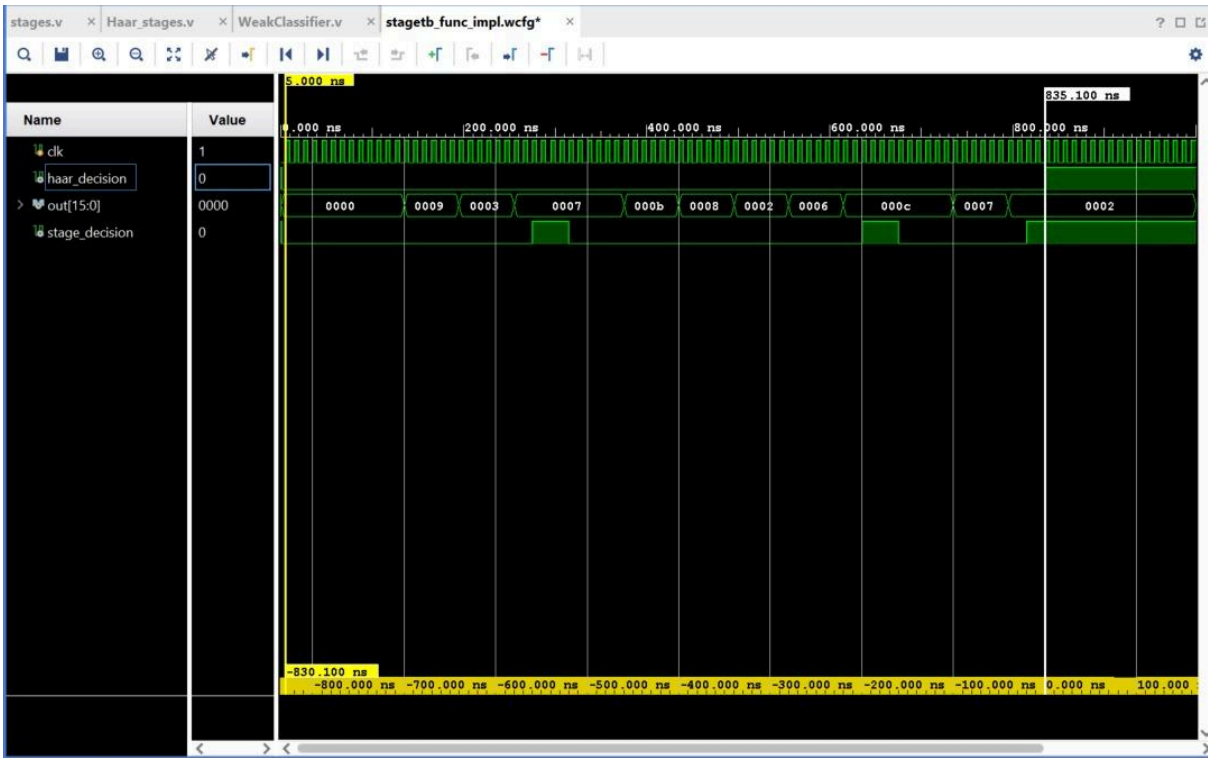
Throughput Calculation:

32% is the max resource utilisation (by block ram)

100 / 32 * Max clock frequency

100 / 32 *51.6* 10 ^ 6 = **161.25* 10^6 operations / second**

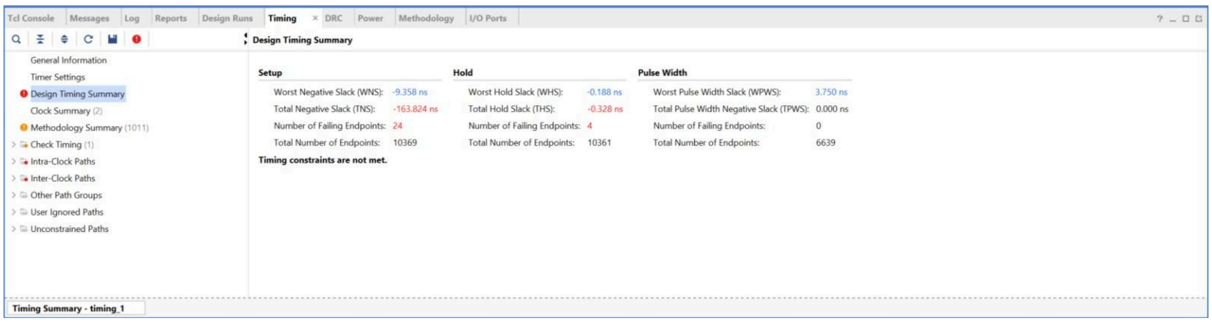
Latency Calculation:



Latency:
((Cursor2 - Cursor1) / 10ns) = 83.10 ns

Maximum Clock Frequency Calculation:
1/T - slack = 51.658 MHz

Slack:



Design Timing Summary

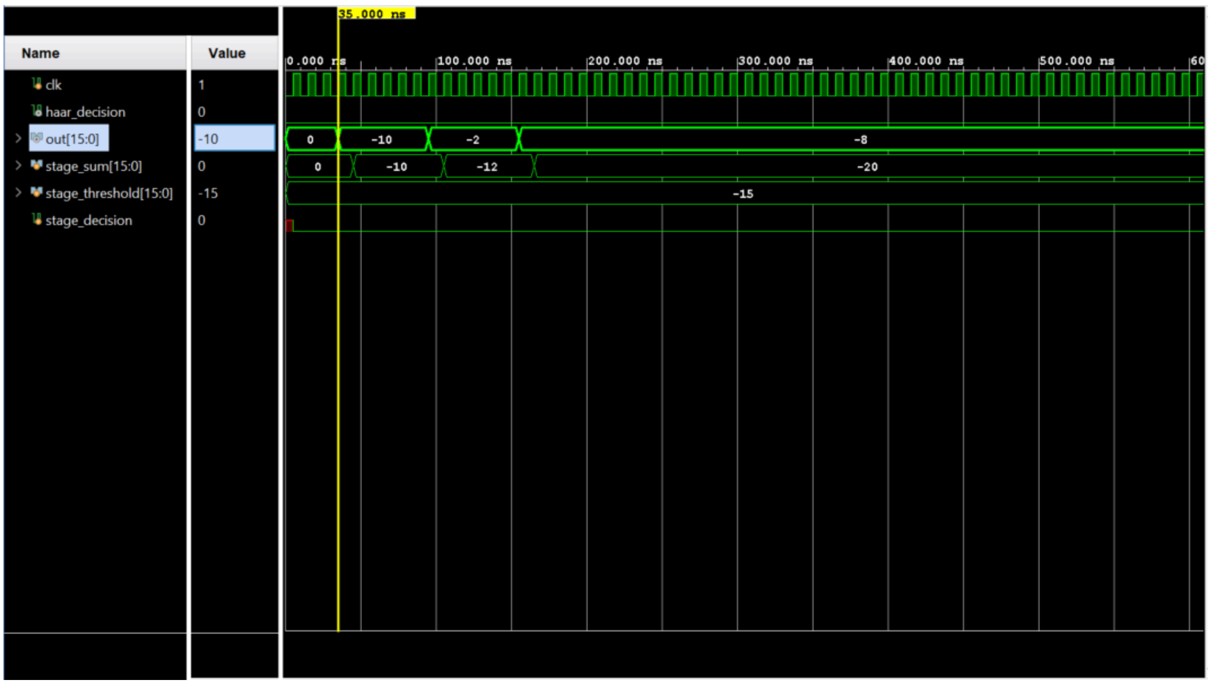
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-9.358 ns	Worst Hold Slack (WHS):	-0.188 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	-163.824 ns	Total Hold Slack (THS):	-0.328 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	24	Number of Failing Endpoints:	4	Number of Failing Endpoints:	0
Total Number of Endpoints:	10369	Total Number of Endpoints:	10361	Total Number of Endpoints:	6639

Timing constraints are not met.

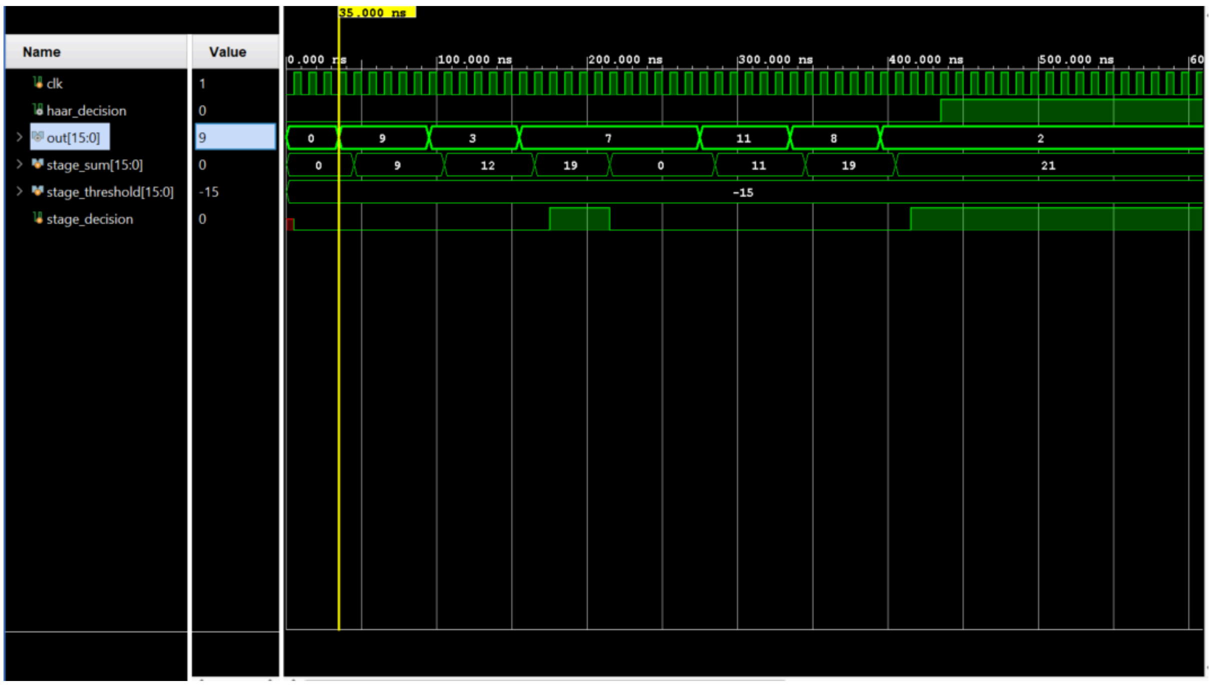
Slack: -9.358ns

SIMULATION OUTPUTS:

#features = 3 in both the stages
CASE1: First stage itself fails:

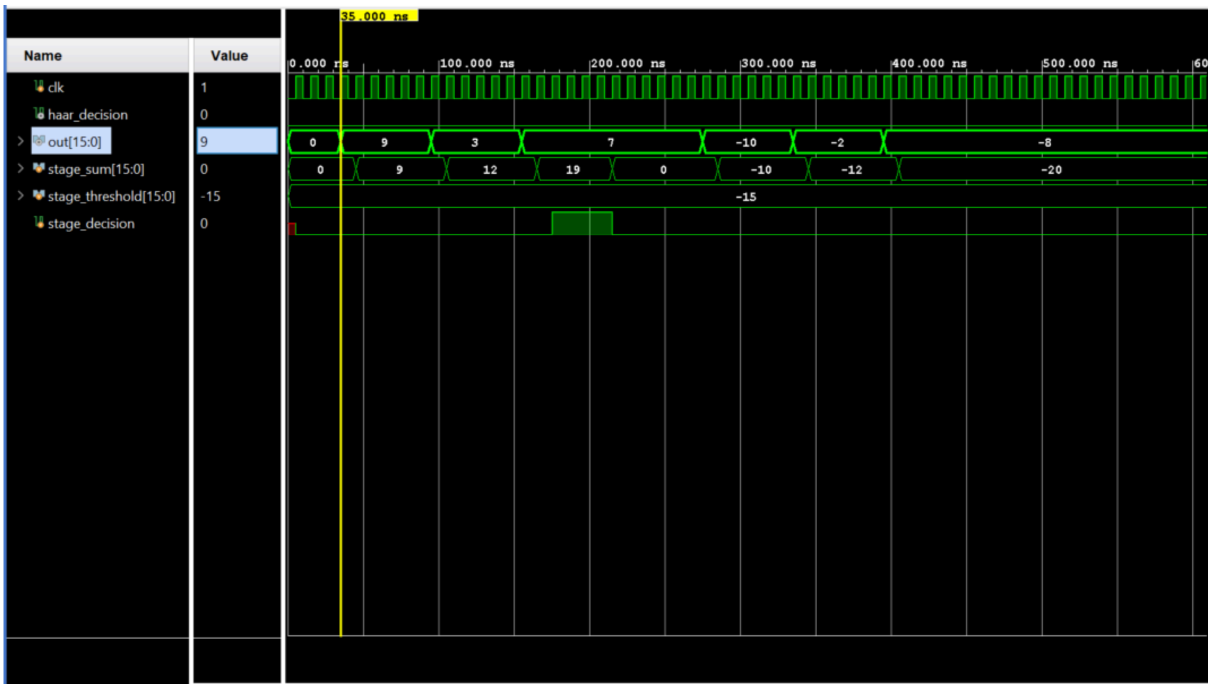


#features = 3 in both the stages
CASE2: Both stages pass:



#features = 3 in both the stages

CASE2: First stage passed, Second stage failed:

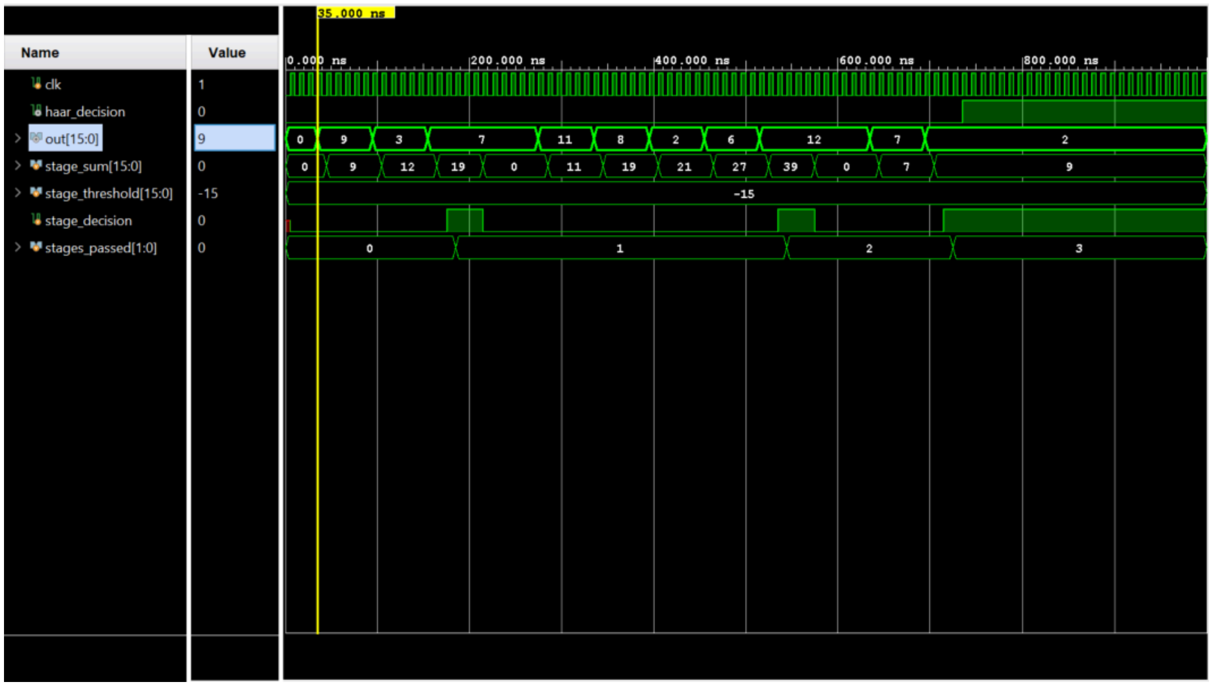


Name	Value
clk	1
haar_decision	0
out[15:0]	9
stage_sum[15:0]	0
stage_threshold[15:0]	-15
stage_decision	0

ALL STAGES PASSED :

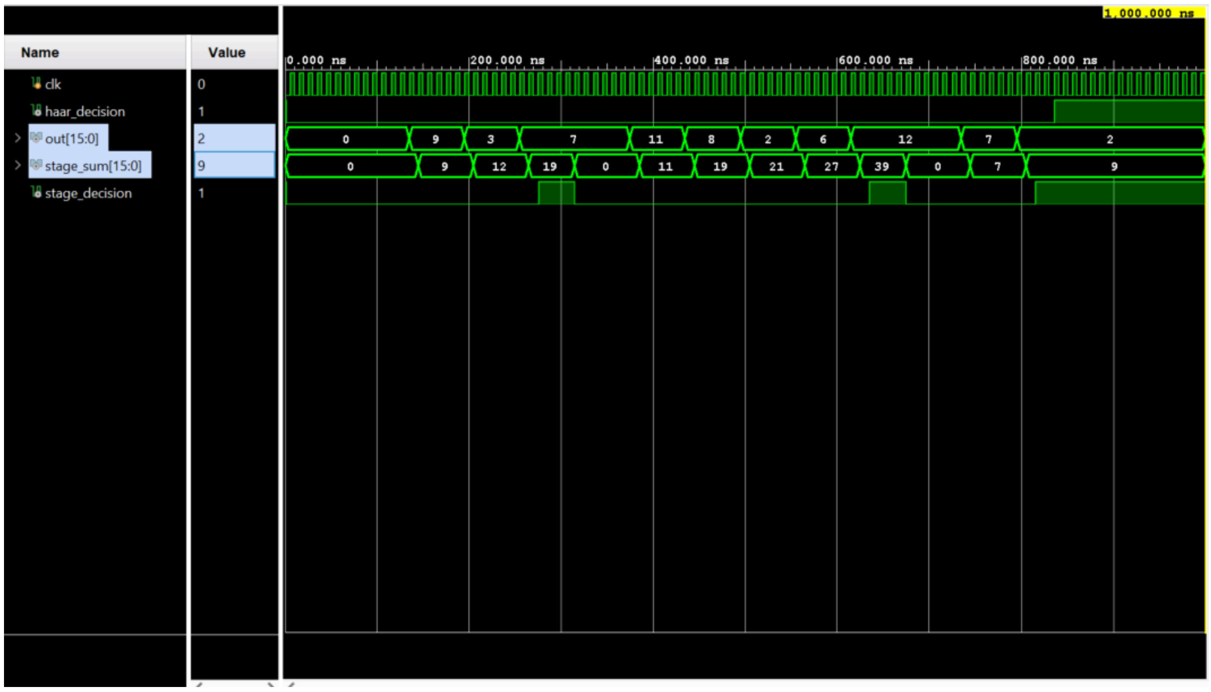
3 STAGE OUTPUT SIMULATION:

BEHAVIORAL SIMULATION:



Name	Value
clk	1
haar_decision	0
out[15:0]	9
stage_sum[15:0]	0
stage_threshold[15:0]	-15
stage_decision	0
stages_passed[1:0]	0

POST-IMPLEMENTATION SIMULATION:



Name	Value
clk	0
haar_decision	1
out[15:0]	2
stage_sum[15:0]	9
stage_decision	1

```
out[15:0]:       0 | 9 | 3 | 7 | 11 | 8 | 2 | 6 | 12 | 7 | 2
stage_sum[15:0]: 0 | 9 | 12 | 19 | 0 | 11 | 19 | 21 | 27 | 39 | 0 | 7 | 9
```

REFERENCES:

We used the below links to read the Haar features and understand about the algorithm:

- https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml
- https://medium.com/@rohan.chaudhury.rc/adaboost-classifier-for-face-detection-using-viola-jones-algorithm-30246527db11
- https://www.analyticsvidhya.com/blog/2022/04/object-detection-using-haar-cascade-opencv/
- https://www.youtube.com/watch?v=hPCTwxF0qf4
- Paper link: https://dl.acm.org/doi/pdf/10.1145/1508128.1508144

  ………