King Fahd University of Petroleum and Minerals

Department of Computer Engineering (COE)
COE 405
Design and Modeling of Digital Systems

High-Speed Parallel HDL Implementation of K-Means Clustering for Image Segmentation

Semester 201

Ahmed Alharbi	201636500
Hamza Alsharif	201642320

Dr Mohammed Elrabaa

Contents

1 Brief Overview

The idea behind our design is that it uses a multi-engine, multi-thread parallel processing approach. Each engine corresponds to 16 threads that represent the means and K threads will be processing and enabled depending on the user's K input. In principle, we do this by only considering the enabled means when clustering the pixels. For our current configuration we are using 10 engines. First we store 5 individual parts of the input image into 10 separate dual-port (we only use the dual read port) memory blocks that each correspond to 2 engines. This allows us to process the whole image in chunks of a 10th of the image size, all in parallel. A thread takes in the pixels as inputs and accumulates their RGB values, but only if that thread is enabled. The accumulated thread values are then sent to the sum unit where they are all added together to get a final accumulated value for the clusters. These resultant centroid accumulated values are then divided by the number of pixels assigned to each of the respective clusters. The division is done through a set of ipcore generated dividers available in Xilinx. The new mean values are then compared with the previous mean values to see if the centroid means have stabilised. In our design we use a maximum allowed difference value of 6, meaning that if the new mean value is within an error range of ±6 of the previous mean value then it is considered to be stable. Once all of the mean values have stabilised the circuit enters the outputting stage. In this stage, the image pixels are compared with the centroid means and the closest mean in terms of Manhattan distance is outputted in place of that pixel. The output pixels are also written to a file. After the outputs have been produced, the circuit is considered to have finished its operation. We believe that this parallel approach will give us a huge boost in performance.

2 Algorithm

The algorithm shown below is a generalised sequential implementation of the K-Means clustering algorithm on hardware. Our actual algorithm involves a module hierarchy and a parallel design which was not easy to represent using sequential-based code. For a more accurate representation of how our circuit works see the ASM diagram.

```
// Note: The algorithm has been written to be closer to verilog-style code even though it does not
// necessarily match its syntax since it is supposed to be pseudo-code

Core { // these operations happen in all of the enabled cores
if (reset) {
    All registers and counters = 0;
    goto Sleep;
}

Sleep:
if (Enable)
    goto ReadPixel;

ReadPixel:
Pixel = PixelIn;
goto CalculateDistance;

```

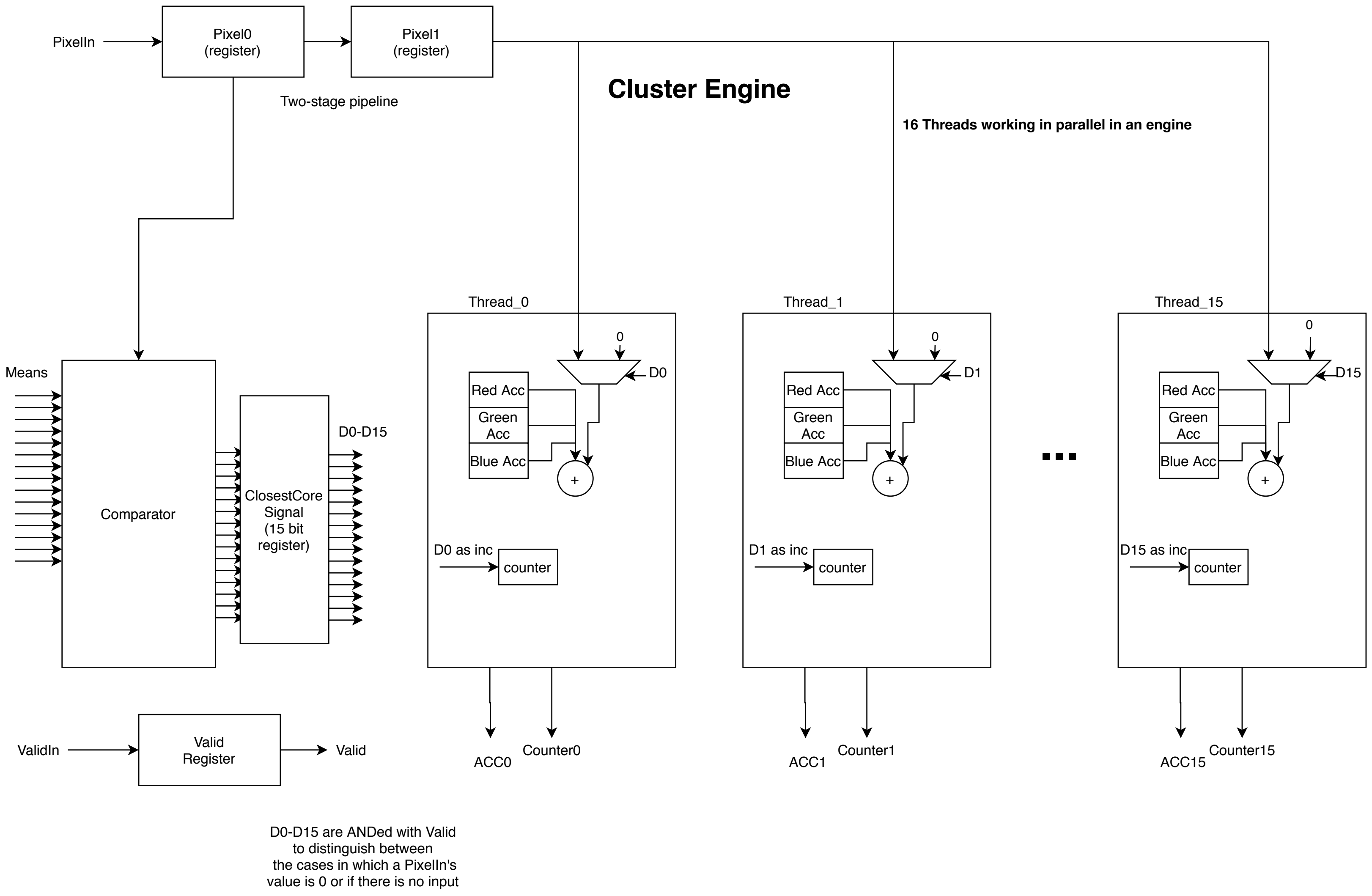
```
CalculateDistance:
Distance = absolute_value(Pixel[23:16] - Mean[23:16]) + absolute_value(Pixel[15:8] - Mean[15:8]) +
absolute_value(Pixel[7:0] - Mean[7:0]);

if (CompareOnly) // if the core only needs to perform a distance comparison without updating its mean value
    goto ReadPixel;
else if (isClosest) // if this core's mean is the closest to the pixel
    goto GroupPixel;
else
    goto Wait;

Wait:
remain idle and wait until the closest core to the pixel to perform its processing;

if (UpdateMean)
    goto UpdateMean;
else
    goto ReadPixel;

GroupPixel:
RedAccumulator = RedAccumulator + Pixel[23:16];
GreenAccumulator = GreenAccumulator + Pixel[15:8];
BlueAccumulator = BlueAccumulator + Pixel[7:0];
PixelCounter = PixelCounter + 1;

if (UpdateMean) // finished reading all pixels and ready to update mean
    goto UpdateMean;
else
    goto ReadPixel;

UpdateMean:
BusyDivide = 1; // this will be a status signal from the divider indicating that it has finished
Mean = RGB Accumulators / PixelCounter; // once the division has finished BusyDivide will become 0

if (!BusyDivide && PreviousMean == Mean)
    MeanStable = 1;
else if (!BusyDivide && PreviousMean != Mean)
    MeanStable = 0;
```

```
goto CoreReady;

CoreReady:
Mean = PreviousMean; // mean value is now constant

if (AllMeansStable) // if all of the means from all cores have stabilized
    goto CoreReady;
else
    goto ReadPixel; // it might be the case that this mean core has stabilized but other mean cores have not
}

System { // This is the overall K-Means system process that uses the 16 cores as components within it
if (reset) {
    All registers and counters = 0;
    goto Initial;
}

Initial:
if (Start)
    goto ReadImageSize;
else
    goto Initial;

ReadImageSize:
ImageSize = SerialIn;
goto ReadK;

ReadK:
K = SerialIn;
goto StoreImage;

StoreImage: // store the image serially from DataIn input
Mem[Address] = DataIn;
Source = 0;

if (Address == ImageSize) {
    Address = 0;
    goto PutPixelOnBus;
}
else {
    Address = Address + 1;
    goto StoreImage;
}

PutPixelOnBus: // finished storing to memory, now we read from it
DataOut = Mem[Address];
Core(DataOut); // cores receive pixels and perform their processing

if (ReadNextPixel for all cores && Address != ImageSize) {// all cores are ready to receive a new pixel
    Address = Address + 1;
    goto PutPixelOnBus;
}
else if (ReadNextPixel for all cores && Address == ImageSize) {
    Address = 0;
    goto PutPixelOnBus;
}
else if (MoveToWrite) {
    Address = 0;
    goto Write;
}
else
    goto PutPixelOnBus;

Write: // writing clusters back into the memory in place of the pixels
Mem[Address] = DataIn;
Source = 1;

if(Address == ImageSize)
    goto Stop;
else {
    Address = Address + 1;
    goto Write;
}

Stop:
Wait for a new image to be received;
```

```
if (NewImage) // signal received that there is a new image
    goto Initial;
else
    goto Stop; // remain here so that the K-Means segmented image remains in the memory;
}
```

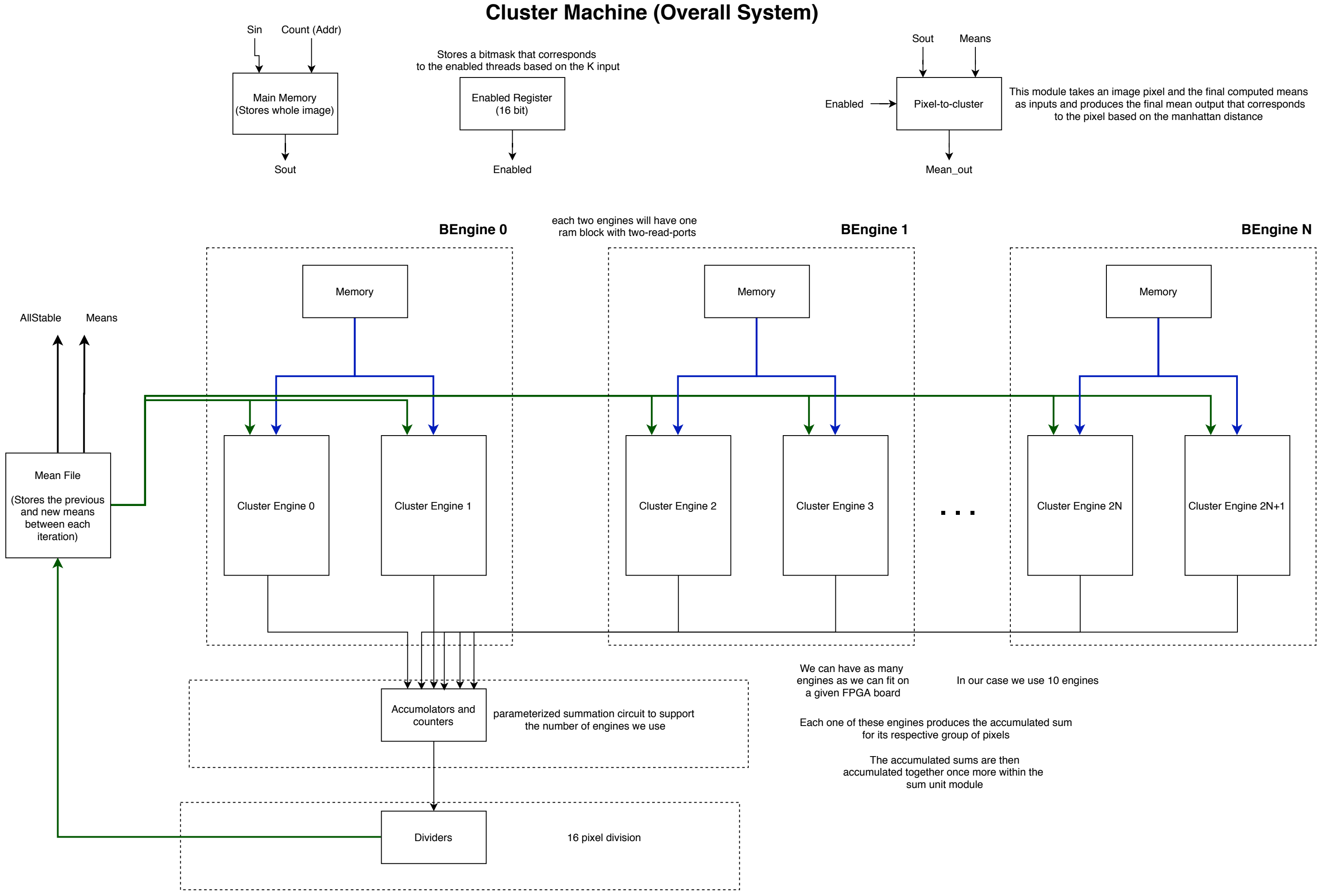
3 Datapath Diagrams

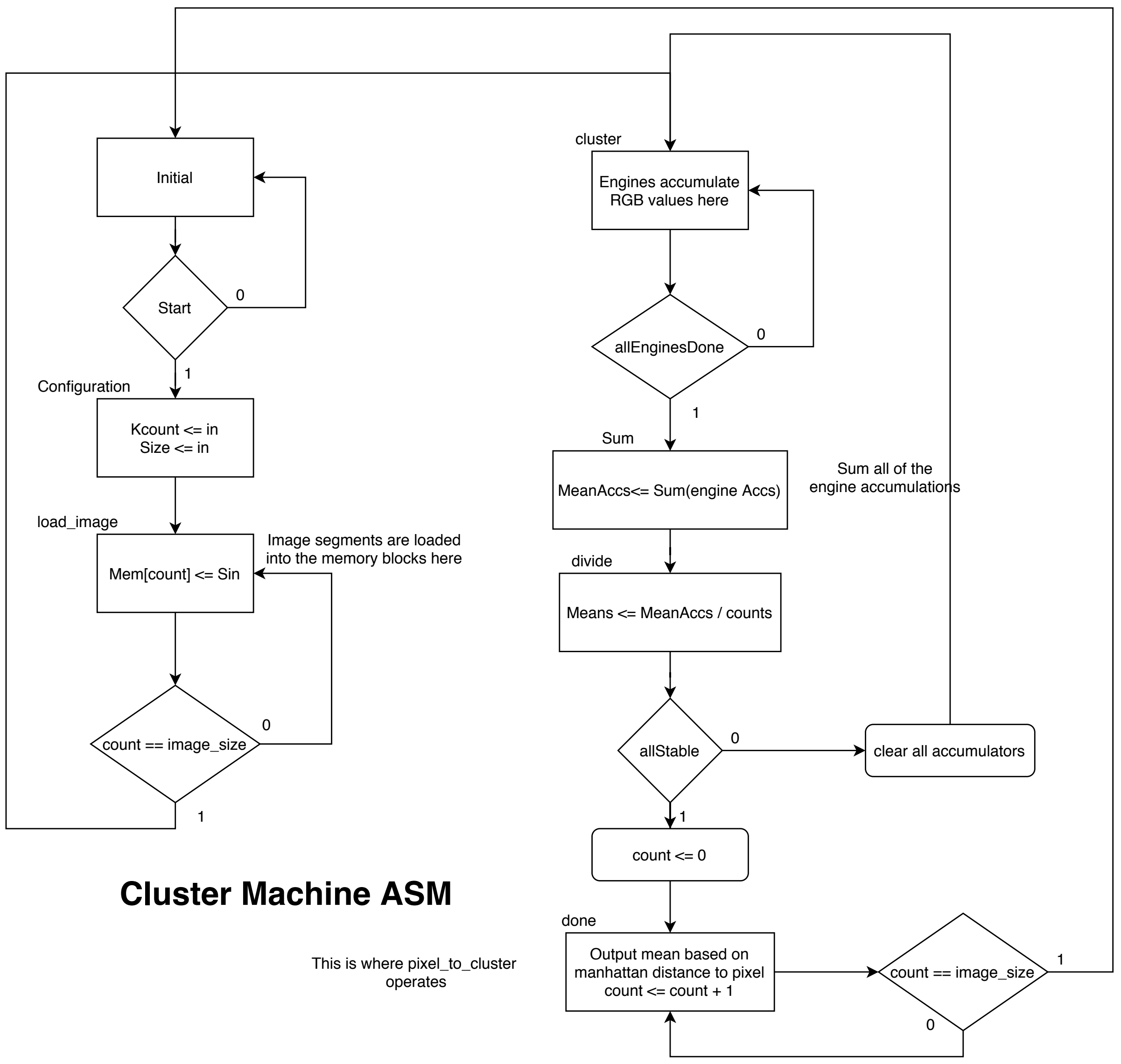
The following pages show the main datapath diagrams for our circuit. First we present the Engine module which is comprised of 16 threads. After that we show the Cluster Machine datapath which is our top level module and is the overall system. For this specific datapath we show a general design that can use N number of Bengines to run 2N engines. The limitation here is how much the FPGA can actually fit in terms of area. For our implementation we use a value of N = 5 for 10 engines as this was the maximum allowable number of engines that fit for our given FPGA.



PixelIn
Pixel0 (register)
Pixel1 (register)
Two-stage pipeline
Cluster Engine
16 Threads working in parallel in an engine
Means
Comparator
ClosestCore Signal (15 bit register)
D0-D15
Thread_0
Thread_1
Thread_15
0
D0
D1
D15
Red Acc
Green Acc
Blue Acc
+
D0 as inc
D1 as inc
D15 as inc
counter
...
ACC0
Counter0
ACC1
Counter1
ACC15
Counter15
ValidIn
Valid Register
Valid
D0-D15 are ANDed with Valid to distinguish between the cases in which a PixelIn's value is 0 or if there is no input

Cluster Machine (Overall System)


Sin
Count (Addr)
Main Memory (Stores whole image)
Sout
Stores a bitmask that corresponds to the enabled threads based on the K input
Enabled Register (16 bit)
Enabled
Sout
Means
Enabled
Pixel-to-cluster
Mean_out
This module takes an image pixel and the final computed means as inputs and produces the final mean output that corresponds to the pixel based on the manhattan distance
BEngine 0
each two engines will have one ram block with two-read-ports
BEngine 1
BEngine N
Memory
Memory
Memory
AllStable
Means
Mean File
(Stores the previous and new means between each iteration)
Cluster Engine 0
Cluster Engine 1
Cluster Engine 2
Cluster Engine 3
. . .
Cluster Engine 2N
Cluster Engine 2N+1
We can have as many engines as we can fit on a given FPGA board
In our case we use 10 engines
Accumolators and counters
parameterized summation circuit to support the number of engines we use
Each one of these engines produces the accumulated sum for its respective group of pixels
The accumulated sums are then accumulated together once more within the sum unit module
Dividers
16 pixel division



Initial
Start
0
1
Configuration
Kcount <= in
Size <= in
load_image
Mem[count] <= Sin
Image segments are loaded into the memory blocks here
count == image_size
0
1
cluster
Engines accumulate RGB values here
allEnginesDone
0
1
Sum
MeanAccs<= Sum(engine Accs)
Sum all of the engine accumulations
divide
Means <= MeanAccs / counts
allStable
0
clear all accumulators
1
count <= 0
done
Output mean based on manhattan distance to pixel count <= count + 1
count == image_size
1
0
This is where pixel_to_cluster operates

Cluster Machine ASM

4 Verilog Modules

4.1 Cluster Machine

As mentioned earlier, the cluster machine is the top-level module and is our parallel implementation of K-Means Clustering for image segmentation in hardware. It contains 5 BEngines that perform the accumulation of the parts of the image. The accumulations from the BEngines are accumulated once more within the sum unit into RGB components for each of the respective clusters. The sum unit accumulations are then divided using a set of dividers and their mean outputs for each iteration is stored in the mean file. The mean file compares the new means with the means from the previous iteration and checks if they are stable, and if all of them are stable then the means computation is complete. Now we output the new segmented image through mean_out using the pixel_to_cluster module. This module compares the image pixels from the main image memory block and outputs the new pixel based on the Manhattan distances of the computed means to the original image pixel. The Cluster Machine also has a control unit as a state machine to synchronize the transitions between the different parallel stages.

4.2 BEngine

A BEngine contains a single dual-port memory block and two engines. Each engine processes half of what is contained of the image in the memory block, so one engine would be operating on the pixels at even addresses and the other would be operating on the pixels at odd addresses.

4.3 Cluster Engine

Cluster engines contain the accumulators and counters for each of the cluster threads for a set of pixels belonging to that engine, as well as the distance comparator and Manhattan modules. These engines cluster all of the RGB values and increment the thread pixel counters based on the Manhattan distances which are compared using the distance comparator module. The engine is two-stage pipelined to reduce the overhead of the Distance Comparator and Manhattan modules.

4.4 Distance Comparator

The distance comparator module takes in the thread distances computed using the Manhattan module and then compares them all through a sequential stage based comparison scheme to find the thread that corresponds to that smallest Manhattan distance. In the first stage we compare 8 distances with 8 other distances and produce the minimum out these 8. These 8 minimums are split into two sets of 4 minimums that are compared with each other in the second stage. In the same way, the resulting 4 minimums are then split into two sets of 2 minimums. The final stage compares these last two minimums and produces a signal based on the final minimum that corresponds to on of the inputs to a priority encoder. The priority encoder outputs the index of the nearest cluster so that a pixel may be accumulated to it within a thread.

4.5 Sum Unit

The sum unit received the accumulated values and counters for all of the cluster threads for all of the engines and accumulates them into a single value for each of the respective means. It does this by sequentially adding the threads using a single accumulator for the RGB values and pixel counts for each of the threads. The output of this module is the final accumulated values for each of the means that are ready for division in the next stage.

4.6 Mean Divider

The mean divider module is simply an encapsulation of 3 dividers corresponding to the RGB values of the pixels. The actual dividers are standard generated Xilinx ipcore dividers. Throughout testing we found that the Xilinx ipcore dividers take 26 cycles to finish 1 division operation. The mean divider module also includes the signal rgb_ready that indicates that all three of the divisions have finished. In the Cluster Machine we use 16 of these mean dividers to divide the cluster accumulators by the number of pixels assigned to the clusters.

4.7 Mean File

The mean file is the module that stores the values of the means. Initially, we store constant values for each mean that are equally spaced between the range of 255 (the max value that a pixel component can have). We store the mean values between each iteration so that we can compare the previous iteration's mean value with that of the new iteration to check for their stability. This is done through the stability checker module, which is a modified comparator that checks to see if the difference of the previous iteration means and the new iteration means is less than 6. The stability checker produces a stable signal for the respective mean if the threshold difference condition is satisfied. If all of the means are stable, then an output signal allStable is produced indicating that the iterations can stop.

4.8 Pixel-to-Cluster

This module is used to output the final segmented pixel values after the means have been computed. This happens in a similar fashion to how the pixels are assigned to each cluster except that in this case we read from the main (whole) image memory block rather than separate engine memory blocks. We also re-use to Manhattan and Distance Comparator modules to be able to assign a mean value to an input pixel. In the test bench, the output mean values are also written to a file so that we may be able to compare our image outputs with the golden model.

5 Results

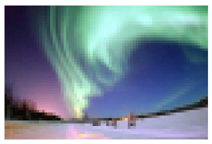


Figure 1: A resized version of the original Wikipedia article image, used for testing.

The final results that we obtained through both simulation and synthesis were highly satisfactory. The algorithm efficiently works as required for different values of K. We tested the algorithm on

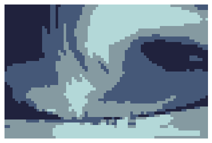


Figure 2: K = 4, Cycles spent processing means = 1701.

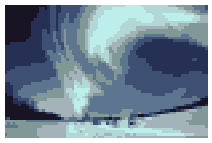


Figure 3: K = 8, Cycles spent processing means = 1276.

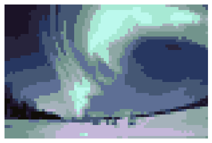


Figure 4: K = 12, Cycles spent processing means = 2127.

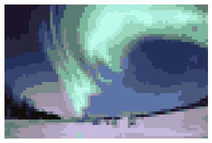
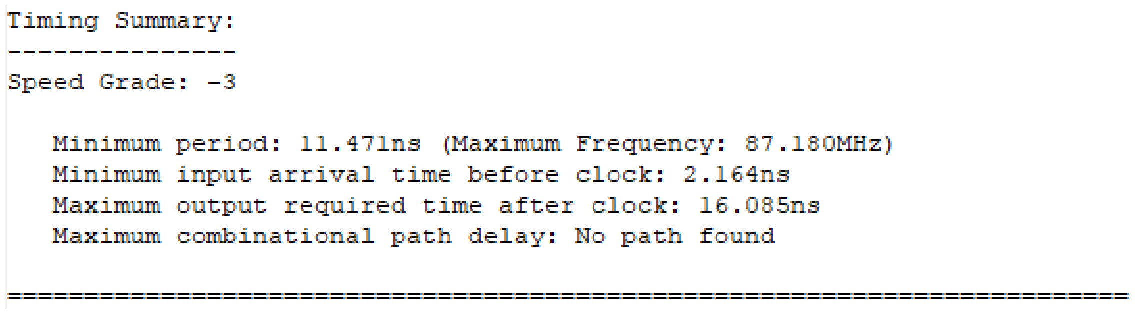


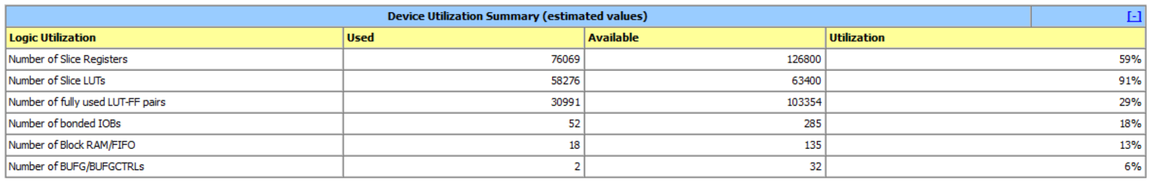
Figure 5: K = 16, Cycles spent processing means = 2551.

a resized version of the image given on the Image Segmentation article on Wikipedia (shown in figure 1) for values of K = 4, 8, 12, and 16 (shown in figures 2, 3, 4, 5). The number of cycles taken for computing 4, 8, 12, and 16 means were 1701, 1276, 2127, and 2551 respectively. It should be noted that the image size was 3800 pixels, so to calculate the total number of cycles you would also have to add 2*3800 to the cycles spent processing because the circuit first spends 3800 cycles to read the image, and then another 3800 cycles to output the segmented image. The circuit also synthesizes and maps on our given board (xc7a100t-3fgg484) and is within the area constraints of the board. The minimum period in which our circuit operates at is 11.471 ns (Max Frequency = 87.180 MHz). The timing report for the circuit is shown below in figure 6. We have also shown that our design meets the area constraints through the device utilization report given in figure 7. For the given clock period, if we compute the total time spent processing it would be clock period * total clock cycles, where the total clock cycles is the sum of the reading, processing, and writing cycles. For our tests above, we can find that for 4, 8, 12, and 16 means the total time for the circuit would be 106.68 μs, 101.82 μs, 111.58 μs, and 116.44 μs respectively. If we only consider the cycles spent processing (neglecting the cycles spent reading and writing), we will find that the total processing times for 4, 8, 12, and 16 means are 19.51 μs, 14.64 μs, 24.40 μs, and 29.26 μs respectively. It must be stated however, that the FPGA choice given to us was disappointing as we believe that we can run at much less clock cycles if we were able to fit more engines on the board, but unfortunately we were limited to using 10. For an in-depth analysis of our module's implementation on an actual board, see the Xilinx reports submitted with this project.



```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 11.471ns (Maximum Frequency: 87.180MHz)
   Minimum input arrival time before clock: 2.164ns
   Maximum output required time after clock: 16.085ns
   Maximum combinational path delay: No path found

=========================================================================
```

Figure 6: Timing Report



Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	76069	126800	59%
Number of Slice LUTs	58276	63400	91%
Number of fully used LUT-FF pairs	30991	103354	29%
Number of bonded IOBs	52	285	18%
Number of Block RAM/FIFO	18	135	13%
Number of BUFG/BUFGCTRLs	2	32	6%

Figure 7: Device Utilization Report